VISIT…

Monographs in Computer Science

Editors

David Gries
Fred B. Schneider

Monographs in Computer Science

Abadi and Cardelli, **A Theory of Objects**

Benosman and Kang [editors], **Panoramic Vision: Sensors, Theory, and Applications**

Bhanu, Lin, Krawiec, **Evolutionary Synthesis of Pattern Recognition Systems**

Broy and Stølen, **Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement**

Brzozowski and Seger, **Asynchronous Circuits**

Burgin, **Super-Recursive Algorithms**

Cantone, Omodeo, and Policriti, **Set Theory for Computing: From Decision Procedures to Declarative Programming with Sets**

Castillo, Gutiérrez, and Hadi, **Expert Systems and Probabilistic Network Models**

Downey and Fellows, **Parameterized Complexity**

Feijen and van Gasteren, **On a Method of Multiprogramming**

Grune and Jacobs, **Parsing Techniques: A Practical Guide, Second Edition**

Herbert and Spärck Jones [editors], **Computer Systems: Theory, Technology, and Applications**

Leiss, **Language Equations**

Levin, Heydon, Mann, and Yu, **Software Configuration Management Using VESTA**

McIver and Morgan [editors], **Programming Methodology**

McIver and Morgan [editors], **Abstraction, Refinement and Proof for Probabilistic Systems**

Misra, **A Discipline of Multiprogramming: Programming Theory for Distributed Applications**

Nielson [editor], **ML with Concurrency**

Paton [editor], **Active Rules in Database Systems**

Poernomo, Crossley, and Wirsing, **Adapting Proof-as-Programs: The Curry-Howard Protocol**

Selig, **Geometrical Methods in Robotics**

Selig, **Geometric Fundamentals of Robotics, Second Edition**

Shasha and Zhu, **High Performance Discovery in Time Series: Techniques and Case Studies**

Tonella and Potrich, **Reverse Engineering of Object Oriented Code**

Dick Grune
Ceriel J.H. Jacobs

Parsing Techniques

A Practical Guide

Second Edition

Springer

Dick Grune and Ceriel J.H. Jacobs
Faculteit Exacte Wetenschappen
Vrije Universiteit
De Boelelaan 1081
1081 HV Amsterdam
The Netherlands

Series Editors
David Gries
Department of Computer Science
Cornell University
4130 Upson Hall
Ithaca, NY 14853-7501
USA

Fred P. Schneider
Department of Computer Science
Cornell University
4130 Upson Hall
Ithaca, NY 14853-7501
USA

ISBN-13: 978-0-387-20248-8 e-ISBN-13: 978-0-387-68954-8

Library of Congress Control Number: 2007936901

©2008 Springer Science+Business Media, LLC

©1990 Ellis Horwood Ltd.

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper. (SB)

9 8 7 6 5 4 3 2 1

springer.com

Preface to the Second Edition

As is fit, this second edition arose out of our readers' demands to read about new developments and our desire to write about them. Although parsing techniques is not a fast moving field, it does move. When the first edition went to press in 1990, there was only one tentative and fairly restrictive algorithm for linear-time substring parsing. Now there are several powerful ones, covering all deterministic languages; we describe them in Chapter 12. In 1990 Theorem 8.1 from a 1961 paper by Bar-Hillel, Perles, and Shamir lay gathering dust; in the last decade it has been used to create new algorithms, and to obtain insight into existing ones. We report on this in Chapter 13.

More and more non-Chomsky systems are used, especially in linguistics. None except two-level grammars had any prominence 20 years ago; we now describe six of them in Chapter 15. Non-canonical parsers were considered oddities for a very long time; now they are among the most powerful linear-time parsers we have; see Chapter 10.

Although still not very practical, marvelous algorithms for parallel parsing have been designed that shed new light on the principles; see Chapter 14. In 1990 a generalized LL parser was deemed impossible; now we describe two in Chapter 11.

Traditionally, and unsurprisingly, parsers have been used for parsing; more recently they are also being used for code generation, data compression and logic language implementation, as shown in Section 17.5. Enough. The reader can find more developments in many places in the book and in the Annotated Bibliography in Chapter 18.

Kees van Reeuwijk has — only half in jest — called our book "a reservation for endangered parsers". We agree — partly; it is more than that — and we make no apologies. Several algorithms in this book have very limited or just no practical value. We have included them because we feel they embody interesting ideas and offer food for thought; they might also grow and acquire practical value. But we also include many algorithms that do have practical value but are sorely underused; describing them here might raise their status in the world.

Exercises and Problems

This book is not a textbook in the school sense of the word. Few universities have a course in Parsing Techniques, and, as stated in the Preface to the First Edition, readers will have very different motivations to use this book. We have therefore included hardly any questions or tasks that exercise the material contained within this book; readers can no doubt make up such tasks for themselves. The questions posed in the problem sections at the end of each chapter usually require the reader to step outside the bounds of the covered material. The problems have been divided into three not too well-defined classes:

- not marked — probably doable in a few minutes to a couple of hours.
- marked *Project* — probably a lot of work, but almost certainly doable.
- marked *Research Project* — almost certainly a lot of work, but hopefully doable.

We make no claims as to the relevance of any of these problems; we hope that some readers will find some of them enlightening, interesting, or perhaps even useful. Ideas, hints, and partial or complete solutions to a number of the problems can be found in Chapter A.

There are also a few questions on formal language that were not answered easily in the existing literature but have some importance to parsing. These have been marked accordingly in the problem sections.

Annotated Bibliography

For the first edition, we, the authors, read and summarized all papers on parsing that we could lay our hands on. Seventeen years later, with the increase in publications and easier access thanks to the Internet, that is no longer possible, much to our chagrin. In the first edition we included all relevant summaries. Again that is not possible now, since doing so would have greatly exceeded the number of pages allotted to this book. The printed version of this second edition includes only those references to the literature and their summaries that are actually referred to in this book. The complete bibliography with summaries as far as available can be found on the web site of this book; it includes its own authors index and subject index. This setup also allows us to list without hesitation technical reports and other material of possibly low accessibility. Often references to sections from Chapter 18 refer to the Web version of those sections; attention is drawn to this by calling them "(Web)Sections".

We do not supply URLs in this book, for two reasons: they are ephemeral and may be incorrect next year, tomorrow, or even before the book is printed; and, especially for software, better URLs may be available by the time you read this book. The best URL is a few well-chosen search terms submitted to a good Web search engine.

Even in the last ten years we have seen a number of Ph.D theses written in languages other than English, specifically German, French, Spanish and Estonian. This choice of language has the regrettable but predictable consequence that their contents have been left out of the main stream of science. This is a loss, both to the authors and to the scientific community. Whether we like it or not, English is the de facto standard language of present-day science. The time that a scientifically in-

terested gentleman of leisure could be expected to read French, German, English, Greek, Latin and a tad of Sanskrit is 150 years in the past; today, students and scientists need the room in their heads and the time in their schedules for the vastly increased amount of knowledge. Although we, the authors, can still read most (but not all) of the above languages and have done our best to represent the contents of the non-English theses adequately, this will not suffice to give them the international attention they deserve.

The Future of Parsing, aka The Crystal Ball

If there will ever be a third edition of this book, we expect it to be substantially thinner (except for the bibliography section!). The reason is that the more parsing algorithms one studies the more they seem similar, and there seems to be great opportunity for unification. Basically almost all parsing is done by top-down search with left-recursion protection; this is true even for traditional bottom-up techniques like LR(1), where the top-down search is built into the LR(1) parse tables. In this respect it is significant that Earley's method is classified as top-down by some and as bottom-up by others. The general memoizing mechanism of tabular parsing takes the exponential sting out of the search. And it seems likely that transforming the usual depth-first search into breadth-first search will yield many of the generalized deterministic algorithms; in this respect we point to Sikkel's Ph.D thesis [158]. Together this seems to cover almost all algorithms in this book, including parsing by intersection. Pure bottom-up parsers without a top-down component are rare and not very powerful.

So in the theoretical future of parsing we see considerable simplification through unification of algorithms; the role that parsing by intersection can play in this is not clear. The simplification does not seem to extend to formal languages: it is still as difficult to prove the intuitively obvious fact that all LL(1) grammars are LR(1) as it was 35 years ago.

The practical future of parsing may lie in advanced pattern recognition, in addition to its traditional tasks; the practical contributions of parsing by intersection are again not clear.

Amsterdam, Amstelveen *Dick Grune*
June 2007 *Ceriel J.H. Jacobs*

Acknowledgments

We thank Manuel E. Bermudez, Stuart Broad, Peter Bumbulis, Salvador Cavadini, Carl Cerecke, Julia Dain, Akim Demaille, Matthew Estes, Wan Fokkink, Brian Ford, Richard Frost, Clemens Grabmayer, Robert Grimm, Karin Harbusch, Stephen Horne, Jaco Imthorn, Quinn Tyler Jackson, Adrian Johnstone, Michiel Koens, Jaroslav Král, Olivier Lecarme, Lillian Lee, Olivier Lefevre, Joop Leo, JianHua Li, Neil Mitchell, Peter Pepper, Wim Pijls, José F. Quesada, Kees van Reeuwijk, Walter L. Ruzzo, Lothar Schmitz, Sylvain Schmitz, Thomas Schoebel-Theuer, Klaas Sikkel, Michael Sperberg-McQueen, Michal Žemlička, Hans Åberg, and many others, for helpful correspondence, comments on and errata to the First Edition, and support for the Second Edition. In particular we want to thank Kees van Reeuwijk and Sylvain Schmitz for their extensive "beta reading", which greatly helped the book — and us.

We thank the Faculteit Exacte Wetenschappen of the Vrije Universiteit for the use of their equipment.

In a wider sense, we extend our thanks to the close to 1500 authors listed in the (Web)Authors Index, who have been so kind as to invent scores of clever and elegant algorithms and techniques for us to exhibit. Every page of this book leans on them.

Preface to the First Edition

Parsing (syntactic analysis) is one of the best understood branches of computer science. Parsers are already being used extensively in a number of disciplines: in computer science (for compiler construction, database interfaces, self-describing databases, artificial intelligence), in linguistics (for text analysis, corpora analysis, machine translation, textual analysis of biblical texts), in document preparation and conversion, in typesetting chemical formulae and in chromosome recognition, to name a few; they can be used (and perhaps are) in a far larger number of disciplines. It is therefore surprising that there is no book which collects the knowledge about parsing and explains it to the non-specialist. Part of the reason may be that parsing has a name for being "difficult". In discussing the Amsterdam Compiler Kit and in teaching compiler construction, it has, however, been our experience that seemingly difficult parsing techniques can be explained in simple terms, given the right approach. The present book is the result of these considerations.

This book does not address a strictly uniform audience. On the contrary, while writing this book, we have consistently tried to imagine giving a course on the subject to a diffuse mixture of students and faculty members of assorted faculties, sophisticated laymen, the avid readers of the science supplement of the large newspapers, etc. Such a course was never given; a diverse audience like that would be too uncoordinated to convene at regular intervals, which is why we wrote this book, to be read, studied, perused or consulted wherever or whenever desired.

Addressing such a varied audience has its own difficulties (and rewards). Although no explicit math was used, it could not be avoided that an amount of mathematical thinking should pervade this book. Technical terms pertaining to parsing have of course been explained in the book, but sometimes a term on the fringe of the subject has been used without definition. Any reader who has ever attended a lecture on a non-familiar subject knows the phenomenon. He skips the term, assumes it refers to something reasonable and hopes it will not recur too often. And then there will be passages where the reader will think we are elaborating the obvious (this paragraph may be one such place). The reader may find solace in the fact that he does not have to doodle his time away or stare out of the window until the lecturer progresses.

On the positive side, and that is the main purpose of this enterprise, we hope that by means of a book with this approach we can reach those who were dimly aware of the existence and perhaps of the usefulness of parsing but who thought it would forever be hidden behind phrases like:

$$\text{Let } \mathfrak{P} \text{ be a mapping } V_N \xrightarrow{\Phi} 2^{(V_N \cup V_T)^*} \text{ and } \mathfrak{H} \text{ a homomorphism } \ldots$$

No knowledge of any particular programming language is required. The book contains two or three programs in Pascal, which serve as actualizations only and play a minor role in the explanation. What is required, though, is an understanding of algorithmic thinking, especially of recursion. Books like *Learning to program* by Howard Johnston (Prentice-Hall, 1985) or *Programming from first principles* by Richard Bornat (Prentice-Hall 1987) provide an adequate background (but supply more detail than required). Pascal was chosen because it is about the only programming language more or less widely available outside computer science environments.

The book features an extensive annotated bibliography. The user of the bibliography is expected to be more than casually interested in parsing and to possess already a reasonable knowledge of it, either through this book or otherwise. The bibliography as a list serves to open up the more accessible part of the literature on the subject to the reader; the annotations are in terse technical prose and we hope they will be useful as stepping stones to reading the actual articles.

On the subject of applications of parsers, this book is vague. Although we suggest a number of applications in Chapter 1, we lack the expertise to supply details. It is obvious that musical compositions possess a structure which can largely be described by a grammar and thus is amenable to parsing, but we shall have to leave it to the musicologists to implement the idea. It was less obvious to us that behaviour at corporate meetings proceeds according to a grammar, but we are told that this is so and that it is a subject of socio-psychological research.

Acknowledgements

We thank the people who helped us in writing this book. Marion de Krieger has retrieved innumerable books and copies of journal articles for us and without her effort the annotated bibliography would be much further from completeness. Ed Keizer has patiently restored peace between us and the pic|tbl|eqn|psfig|troff pipeline, on the many occasions when we abused, overloaded or just plainly misunderstood the latter. Leo van Moergestel has made the hardware do things for us that it would not do for the uninitiated. We also thank Erik Baalbergen, Frans Kaashoek, Erik Groeneveld, Gerco Ballintijn, Jaco Imthorn, and Egon Amada for their critical remarks and contributions. The rose at the end of Chapter 2 is by Arwen Grune. Ilana and Lily Grune typed parts of the text on various occasions.

We thank the Faculteit Wiskunde en Informatica of the Vrije Universiteit for the use of the equipment.

In a wider sense, we extend our thanks to the hundreds of authors who have been so kind as to invent scores of clever and elegant algorithms and techniques for us to exhibit. We hope we have named them all in our bibliography.

Amsterdam, Amstelveen *Dick Grune*
July 1990 *Ceriel J.H. Jacobs*

Contents

Preface to the Second Edition v

Preface to the First Edition xi

1 Introduction 1
1.1 Parsing as a Craft 2
1.2 The Approach Used 2
1.3 Outline of the Contents 3
1.4 The Annotated Bibliography 4

2 Grammars as a Generating Device 5
2.1 Languages as Infinite Sets 5
2.1.1 Language 5
2.1.2 Grammars 7
2.1.3 Problems with Infinite Sets 8
2.1.4 Describing a Language through a Finite Recipe 12
2.2 Formal Grammars 14
2.2.1 The Formalism of Formal Grammars 14
2.2.2 Generating Sentences from a Formal Grammar 15
2.2.3 The Expressive Power of Formal Grammars 17
2.3 The Chomsky Hierarchy of Grammars and Languages 19
2.3.1 Type 1 Grammars 19
2.3.2 Type 2 Grammars 23
2.3.3 Type 3 Grammars 30
2.3.4 Type 4 Grammars 33
2.3.5 Conclusion 34
2.4 Actually Generating Sentences from a Grammar 34
2.4.1 The Phrase-Structure Case 34
2.4.2 The CS Case 36
2.4.3 The CF Case 36
2.5 To Shrink or Not To Shrink 38

2.6 Grammars that Produce the Empty Language 41
2.7 The Limitations of CF and FS Grammars 42
2.7.1 The *uvwxy* Theorem 42
2.7.2 The *uvw* Theorem 45
2.8 CF and FS Grammars as Transition Graphs 45
2.9 Hygiene in Context-Free Grammars 47
2.9.1 Undefined Non-Terminals 48
2.9.2 Unreachable Non-Terminals 48
2.9.3 Non-Productive Rules and Non-Terminals 48
2.9.4 Loops 48
2.9.5 Cleaning up a Context-Free Grammar 49
2.10 Set Properties of Context-Free and Regular Languages 52
2.11 The Semantic Connection 54
2.11.1 Attribute Grammars 54
2.11.2 Transduction Grammars 55
2.11.3 Augmented Transition Networks 56
2.12 A Metaphorical Comparison of Grammar Types 56
2.13 Conclusion 59

3 Introduction to Parsing 61
3.1 The Parse Tree 61
3.1.1 The Size of a Parse Tree 62
3.1.2 Various Kinds of Ambiguity 63
3.1.3 Linearization of the Parse Tree 65
3.2 Two Ways to Parse a Sentence 65
3.2.1 Top-Down Parsing 66
3.2.2 Bottom-Up Parsing 67
3.2.3 Applicability 68
3.3 Non-Deterministic Automata 69
3.3.1 Constructing the NDA 70
3.3.2 Constructing the Control Mechanism 70
3.4 Recognition and Parsing for Type 0 to Type 4 Grammars 71
3.4.1 Time Requirements 71
3.4.2 Type 0 and Type 1 Grammars 72
3.4.3 Type 2 Grammars 73
3.4.4 Type 3 Grammars 75
3.4.5 Type 4 Grammars 75
3.5 An Overview of Context-Free Parsing Methods 76
3.5.1 Directionality 76
3.5.2 Search Techniques 77
3.5.3 General Directional Methods 78
3.5.4 Linear Methods 80
3.5.5 Deterministic Top-Down and Bottom-Up Methods 82
3.5.6 Non-Canonical Methods 83
3.5.7 Generalized Linear Methods 84

3.5.8 Conclusion 84
3.6 The "Strength" of a Parsing Technique 84
3.7 Representations of Parse Trees 85
3.7.1 Parse Trees in the Producer-Consumer Model 86
3.7.2 Parse Trees in the Data Structure Model 87
3.7.3 Parse Forests 87
3.7.4 Parse-Forest Grammars 91
3.8 When are we done Parsing? 93
3.9 Transitive Closure 95
3.10 The Relation between Parsing and Boolean Matrix Multiplication . . 97
3.11 Conclusion 100

4 General Non-Directional Parsing 103
4.1 Unger's Parsing Method 104
4.1.1 Unger's Method without ε-Rules or Loops 104
4.1.2 Unger's Method with ε-Rules 107
4.1.3 Getting Parse-Forest Grammars from Unger Parsing 110
4.2 The CYK Parsing Method 112
4.2.1 CYK Recognition with General CF Grammars 112
4.2.2 CYK Recognition with a Grammar in Chomsky Normal Form 116
4.2.3 Transforming a CF Grammar into Chomsky Normal Form . . 119
4.2.4 The Example Revisited 122
4.2.5 CYK Parsing with Chomsky Normal Form 124
4.2.6 Undoing the Effect of the CNF Transformation 125
4.2.7 A Short Retrospective of CYK 128
4.2.8 Getting Parse-Forest Grammars from CYK Parsing 129
4.3 Tabular Parsing 129
4.3.1 Top-Down Tabular Parsing 131
4.3.2 Bottom-Up Tabular Parsing 133
4.4 Conclusion 134

5 Regular Grammars and Finite-State Automata 137
5.1 Applications of Regular Grammars 137
5.1.1 Regular Languages in CF Parsing 137
5.1.2 Systems with Finite Memory 139
5.1.3 Pattern Searching 141
5.1.4 SGML and XML Validation 141
5.2 Producing from a Regular Grammar 141
5.3 Parsing with a Regular Grammar 143
5.3.1 Replacing Sets by States 144
5.3.2 ε-Transitions and Non-Standard Notation 147
5.4 Manipulating Regular Grammars and Regular Expressions 148
5.4.1 Regular Grammars from Regular Expressions 149
5.4.2 Regular Expressions from Regular Grammars 151
5.5 Manipulating Regular Languages 152

5.6 Left-Regular Grammars 154
5.7 Minimizing Finite-State Automata 156
5.8 Top-Down Regular Expression Recognition 158
5.8.1 The Recognizer 158
5.8.2 Evaluation 159
5.9 Semantics in FS Systems 160
5.10 Fast Text Search Using Finite-State Automata 161
5.11 Conclusion 162

6 General Directional Top-Down Parsing 165
6.1 Imitating Leftmost Derivations 165
6.2 The Pushdown Automaton 167
6.3 Breadth-First Top-Down Parsing 171
6.3.1 An Example 173
6.3.2 A Counterexample: Left Recursion 173
6.4 Eliminating Left Recursion 175
6.5 Depth-First (Backtracking) Parsers 176
6.6 Recursive Descent 177
6.6.1 A Naive Approach 179
6.6.2 Exhaustive Backtracking Recursive Descent 183
6.6.3 Breadth-First Recursive Descent 185
6.7 Definite Clause Grammars 188
6.7.1 Prolog 188
6.7.2 The DCG Format 189
6.7.3 Getting Parse Tree Information 190
6.7.4 Running Definite Clause Grammar Programs 190
6.8 Cancellation Parsing 192
6.8.1 Cancellation Sets 192
6.8.2 The Transformation Scheme 193
6.8.3 Cancellation Parsing with ε-Rules 196
6.9 Conclusion 197

7 General Directional Bottom-Up Parsing 199
7.1 Parsing by Searching 201
7.1.1 Depth-First (Backtracking) Parsing 201
7.1.2 Breadth-First (On-Line) Parsing 202
7.1.3 A Combined Representation 203
7.1.4 A Slightly More Realistic Example 204
7.2 The Earley Parser 206
7.2.1 The Basic Earley Parser 206
7.2.2 The Relation between the Earley and CYK Algorithms 212
7.2.3 Handling ε-Rules 214
7.2.4 Exploiting Look-Ahead 219
7.2.5 Left and Right Recursion 224
7.3 Chart Parsing 226

7.3.1 Inference Rules . . . 227
7.3.2 A Transitive Closure Algorithm . . . 227
7.3.3 Completion . . . 229
7.3.4 Bottom-Up (Actually Left-Corner) . . . 229
7.3.5 The Agenda . . . 229
7.3.6 Top-Down . . . 231
7.3.7 Conclusion . . . 232
7.4 Conclusion . . . 233

8 Deterministic Top-Down Parsing . . . 235
8.1 Replacing Search by Table Look-Up . . . 236
8.2 LL(1) Parsing . . . 239
8.2.1 LL(1) Parsing without ε-Rules . . . 239
8.2.2 LL(1) Parsing with ε-Rules . . . 242
8.2.3 LL(1) versus Strong-LL(1) . . . 247
8.2.4 Full LL(1) Parsing . . . 248
8.2.5 Solving LL(1) Conflicts . . . 251
8.2.6 LL(1) and Recursive Descent . . . 253
8.3 Increasing the Power of Deterministic LL Parsing . . . 254
8.3.1 LL(k) Grammars . . . 254
8.3.2 Linear-Approximate LL(k) . . . 256
8.3.3 LL-Regular . . . 257
8.4 Getting a Parse Tree Grammar from LL(1) Parsing . . . 258
8.5 Extended LL(1) Grammars . . . 259
8.6 Conclusion . . . 260

9 Deterministic Bottom-Up Parsing . . . 263
9.1 Simple Handle-Finding Techniques . . . 265
9.2 Precedence Parsing . . . 266
9.2.1 Parenthesis Generators . . . 267
9.2.2 Constructing the Operator-Precedence Table . . . 269
9.2.3 Precedence Functions . . . 271
9.2.4 Further Precedence Methods . . . 272
9.3 Bounded-Right-Context Parsing . . . 275
9.3.1 Bounded-Context Techniques . . . 276
9.3.2 Floyd Productions . . . 277
9.4 LR Methods . . . 278
9.5 LR(0) . . . 280
9.5.1 The LR(0) Automaton . . . 280
9.5.2 Using the LR(0) Automaton . . . 283
9.5.3 LR(0) Conflicts . . . 286
9.5.4 ε-LR(0) Parsing . . . 287
9.5.5 Practical LR Parse Table Construction . . . 289
9.6 LR(1) . . . 290
9.6.1 LR(1) with ε-Rules . . . 295

9.6.2 LR($k > 1$) Parsing . . . 297
9.6.3 Some Properties of LR(k) Parsing . . . 299
9.7 LALR(1) . . . 300
9.7.1 Constructing the LALR(1) Parsing Tables . . . 302
9.7.2 Identifying LALR(1) Conflicts . . . 314
9.8 SLR(1) . . . 314
9.9 Conflict Resolvers . . . 315
9.10 Further Developments of LR Methods . . . 316
9.10.1 Elimination of Unit Rules . . . 316
9.10.2 Reducing the Stack Activity . . . 317
9.10.3 Regular Right Part Grammars . . . 318
9.10.4 Incremental Parsing . . . 318
9.10.5 Incremental Parser Generation . . . 318
9.10.6 Recursive Ascent . . . 319
9.10.7 Regular Expressions of LR Languages . . . 319
9.11 Getting a Parse Tree Grammar from LR Parsing . . . 319
9.12 Left and Right Contexts of Parsing Decisions . . . 320
9.12.1 The Left Context of a State . . . 321
9.12.2 The Right Context of an Item . . . 322
9.13 Exploiting the Left and Right Contexts . . . 323
9.13.1 Discriminating-Reverse (DR) Parsing . . . 324
9.13.2 LR-Regular . . . 327
9.13.3 LAR(m) Parsing . . . 333
9.14 LR(k) as an Ambiguity Test . . . 338
9.15 Conclusion . . . 338

10 Non-Canonical Parsers . . . 343
10.1 Top-Down Non-Canonical Parsing . . . 344
10.1.1 Left-Corner Parsing . . . 344
10.1.2 Deterministic Cancellation Parsing . . . 353
10.1.3 Partitioned LL . . . 354
10.1.4 Discussion . . . 357
10.2 Bottom-Up Non-Canonical Parsing . . . 357
10.2.1 Total Precedence . . . 358
10.2.2 NSLR(1) . . . 359
10.2.3 LR(k,∞) . . . 364
10.2.4 Partitioned LR . . . 372
10.3 General Non-Canonical Parsing . . . 377
10.4 Conclusion . . . 379

11 Generalized Deterministic Parsers . . . 381
11.1 Generalized LR Parsing . . . 382
11.1.1 The Basic GLR Parsing Algorithm . . . 382
11.1.2 Necessary Optimizations . . . 383
11.1.3 Hidden Left Recursion and Loops . . . 387

11.1.4 Extensions and Improvements 390
11.2 Generalized LL Parsing 391
11.2.1 Simple Generalized LL Parsing 391
11.2.2 Generalized LL Parsing with Left-Recursion 393
11.2.3 Generalized LL Parsing with ε-Rules 395
11.2.4 Generalized Cancellation and LC Parsing 397
11.3 Conclusion 398

12 Substring Parsing 399
12.1 The Suffix Grammar 401
12.2 General (Non-Linear) Methods 402
12.2.1 A Non-Directional Method 403
12.2.2 A Directional Method 407
12.3 Linear-Time Methods for LL and LR Grammars 408
12.3.1 Linear-Time Suffix Parsing for LL(1) Grammars 409
12.3.2 Linear-Time Suffix Parsing for LR(1) Grammars 414
12.3.3 Tabular Methods 418
12.3.4 Discussion 421
12.4 Conclusion 421

13 Parsing as Intersection 425
13.1 The Intersection Algorithm 426
13.1.1 The Rule Sets I_{rules}, I_{rough}, and I 427
13.1.2 The Languages of I_{rules}, I_{rough}, and I 429
13.1.3 An Example: Parsing Arithmetic Expressions 430
13.2 The Parsing of FSAs 431
13.2.1 Unknown Tokens 431
13.2.2 Substring Parsing by Intersection 431
13.2.3 Filtering 435
13.3 Time and Space Requirements 436
13.4 Reducing the Intermediate Size: Earley's Algorithm on FSAs 437
13.5 Error Handling Using Intersection Parsing 439
13.6 Conclusion 441

14 Parallel Parsing 443
14.1 The Reasons for Parallel Parsing 443
14.2 Multiple Serial Parsers 444
14.3 Process-Configuration Parsers 447
14.3.1 A Parallel Bottom-up GLR Parser 448
14.3.2 Some Other Process-Configuration Parsers 452
14.4 Connectionist Parsers 453
14.4.1 Boolean Circuits 453
14.4.2 A CYK Recognizer on a Boolean Circuit 454
14.4.3 Rytter's Algorithm 460
14.5 Conclusion 470

15 Non-Chomsky Grammars and Their Parsers 473
15.1 The Unsuitability of Context-Sensitive Grammars 473
15.1.1 Understanding Context-Sensitive Grammars 474
15.1.2 Parsing with Context-Sensitive Grammars 475
15.1.3 Expressing Semantics in Context-Sensitive Grammars 475
15.1.4 Error Handling in Context-Sensitive Grammars 475
15.1.5 Alternatives 476
15.2 Two-Level Grammars 476
15.2.1 VW Grammars 477
15.2.2 Expressing Semantics in a VW Grammar 480
15.2.3 Parsing with VW Grammars 482
15.2.4 Error Handling in VW Grammars 484
15.2.5 Infinite Symbol Sets 484
15.3 Attribute and Affix Grammars 485
15.3.1 Attribute Grammars 485
15.3.2 Affix Grammars 488
15.4 Tree-Adjoining Grammars 492
15.4.1 Cross-Dependencies 492
15.4.2 Parsing with TAGs 497
15.5 Coupled Grammars 500
15.5.1 Parsing with Coupled Grammars 501
15.6 Ordered Grammars 502
15.6.1 Rule Ordering by Control Grammar 502
15.6.2 Parsing with Rule-Ordered Grammars 503
15.6.3 Marked Ordered Grammars 504
15.6.4 Parsing with Marked Ordered Grammars 505
15.7 Recognition Systems 506
15.7.1 Properties of a Recognition System 507
15.7.2 Implementing a Recognition System 509
15.7.3 Parsing with Recognition Systems 512
15.7.4 Expressing Semantics in Recognition Systems 512
15.7.5 Error Handling in Recognition Systems 513
15.8 Boolean Grammars 514
15.8.1 Expressing Context Checks in Boolean Grammars 514
15.8.2 Parsing with Boolean Grammars 516
15.8.3 §-Calculus 516
15.9 Conclusion 517

16 Error Handling 521
16.1 Detection versus Recovery versus Correction 521
16.2 Parsing Techniques and Error Detection 523
16.2.1 Error Detection in Non-Directional Parsing Methods 523
16.2.2 Error Detection in Finite-State Automata 524
16.2.3 Error Detection in General Directional Top-Down Parsers . . 524
16.2.4 Error Detection in General Directional Bottom-Up Parsers . . 524

16.2.5 Error Detection in Deterministic Top-Down Parsers 525
16.2.6 Error Detection in Deterministic Bottom-Up Parsers 525
16.3 Recovering from Errors 526
16.4 Global Error Handling 526
16.5 Regional Error Handling 530
16.5.1 Backward/Forward Move Error Recovery 530
16.5.2 Error Recovery with Bounded-Context Grammars 532
16.6 Local Error Handling 533
16.6.1 Panic Mode 534
16.6.2 FOLLOW-Set Error Recovery 534
16.6.3 Acceptable-Sets Derived from Continuations 535
16.6.4 Insertion-Only Error Correction 537
16.6.5 Locally Least-Cost Error Recovery 539
16.7 Non-Correcting Error Recovery 540
16.7.1 Detection and Recovery 540
16.7.2 Locating the Error 541
16.8 Ad Hoc Methods 542
16.8.1 Error Productions 542
16.8.2 Empty Table Slots 543
16.8.3 Error Tokens 543
16.9 Conclusion 543

17 Practical Parser Writing and Usage 545
17.1 A Comparative Survey 545
17.1.1 Considerations 545
17.1.2 General Parsers 546
17.1.3 General Substring Parsers 547
17.1.4 Linear-Time Parsers 548
17.1.5 Linear-Time Substring Parsers 549
17.1.6 Obtaining and Using a Parser Generator 549
17.2 Parser Construction 550
17.2.1 Interpretive, Table-Based, and Compiled Parsers 550
17.2.2 Parsing Methods and Implementations 551
17.3 A Simple General Context-Free Parser 553
17.3.1 Principles of the Parser 553
17.3.2 The Program 554
17.3.3 Handling Left Recursion 559
17.3.4 Parsing in Polynomial Time 560
17.4 Programming Language Paradigms 563
17.4.1 Imperative and Object-Oriented Programming 563
17.4.2 Functional Programming 564
17.4.3 Logic Programming 567
17.5 Alternative Uses of Parsing 567
17.5.1 Data Compression 567
17.5.2 Machine Code Generation 570

17.5.3 Support of Logic Languages 573
17.6 Conclusion 573

18 Annotated Bibliography 575
18.1 Major Parsing Subjects 576
18.1.1 Unrestricted PS and CS Grammars 576
18.1.2 General Context-Free Parsing 576
18.1.3 LL Parsing 584
18.1.4 LR Parsing 585
18.1.5 Left-Corner Parsing 592
18.1.6 Precedence and Bounded-Right-Context Parsing 593
18.1.7 Finite-State Automata 596
18.1.8 General Books and Papers on Parsing 599
18.2 Advanced Parsing Subjects 601
18.2.1 Generalized Deterministic Parsing 601
18.2.2 Non-Canonical Parsing 605
18.2.3 Substring Parsing 609
18.2.4 Parsing as Intersection 611
18.2.5 Parallel Parsing Techniques 612
18.2.6 Non-Chomsky Systems 614
18.2.7 Error Handling 623
18.2.8 Incremental Parsing 629
18.3 Parsers and Applications 630
18.3.1 Parser Writing 630
18.3.2 Parser-Generating Systems 634
18.3.3 Applications 634
18.3.4 Parsing and Deduction 635
18.3.5 Parsing Issues in Natural Language Handling 636
18.4 Support Material 638
18.4.1 Formal Languages 638
18.4.2 Approximation Techniques 641
18.4.3 Transformations on Grammars 641
18.4.4 Miscellaneous Literature 642

A Hints and Solutions to Selected Problems 645

Author Index 651

Subject Index 655

1

Introduction

Parsing is the process of structuring a linear representation in accordance with a given grammar. This definition has been kept abstract on purpose to allow as wide an interpretation as possible. The "linear representation" may be a sentence, a computer program, a knitting pattern, a sequence of geological strata, a piece of music, actions in ritual behavior, in short any linear sequence in which the preceding elements in some way restrict[1] the next element. For some of the examples the grammar is well known, for some it is an object of research, and for some our notion of a grammar is only just beginning to take shape.

For each grammar, there are generally an infinite number of linear representations ("sentences") that can be structured with it. That is, a finite-size grammar can supply structure to an infinite number of sentences. This is the main strength of the grammar paradigm and indeed the main source of the importance of grammars: they summarize succinctly the structure of an infinite number of objects of a certain class.

There are several reasons to perform this structuring process called parsing. One reason derives from the fact that the obtained structure helps us to process the object further. When we know that a certain segment of a sentence is the subject, that information helps in understanding or translating the sentence. Once the structure of a document has been brought to the surface, it can be converted more easily.

A second reason is related to the fact that the grammar in a sense represents our understanding of the observed sentences: the better a grammar we can give for the movements of bees, the deeper our understanding is of them.

A third lies in the completion of missing information that parsers, and especially error-repairing parsers, can provide. Given a reasonable grammar of the language, an error-repairing parser can suggest possible word classes for missing or unknown words on clay tablets.

The reverse problem — given a (large) set of sentences, find the/a grammar which produces them — is called *grammatical inference*. Much less is known about it than about parsing, but progress is being made. The subject would require a complete

[1] If there is no restriction, the sequence still has a grammar, but this grammar is trivial and uninformative.

book. Proceedings of the International Colloquiums on Grammatical Inference are published as Lecture Notes in Artificial Intelligence by Springer.

1.1 Parsing as a Craft

Parsing is no longer an arcane art; it has not been so since the early 1970s when Aho, Ullman, Knuth and many others put various parsing techniques solidly on their theoretical feet. It need not be a mathematical discipline either; the inner workings of a parser can be visualized, understood and modified to fit the application, with little more than cutting and pasting strings.

There is a considerable difference between a mathematician's view of the world and a computer scientist's. To a mathematician all structures are static: they have always been and will always be; the only time dependence is that we just have not discovered them all yet. The computer scientist is concerned with (and fascinated by) the continuous creation, combination, separation and destruction of structures: time is of the essence. In the hands of a mathematician, the Peano axioms create the integers without reference to time, but if a computer scientist uses them to implement integer addition, he finds they describe a very slow process, which is why he will be looking for a more efficient approach. In this respect the computer scientist has more in common with the physicist and the chemist; like them, he cannot do without a solid basis in several branches of applied mathematics, but, like them, he is willing (and often virtually obliged) to take on faith certain theorems handed to him by the mathematician. Without the rigor of mathematics all science would collapse, but not all inhabitants of a building need to know all the spars and girders that keep it upright. Factoring out certain detailed knowledge to specialists reduces the intellectual complexity of a task, which is one of the things computer science is about.

This is the vein in which this book is written: parsing for anybody who has parsing to do: the compiler writer, the linguist, the database interface writer, the geologist or musicologist who wants to test grammatical descriptions of their respective objects of interest, and so on. We require a good ability to visualize, some programming experience and the willingness and patience to follow non-trivial examples; there is nothing better for understanding a kangaroo than seeing it jump. We treat, of course, the popular parsing techniques, but we will not shun some weird techniques that look as if they are of theoretical interest only: they often offer new insights and a reader might find an application for them.

1.2 The Approach Used

This book addresses the reader at least three different levels. The interested non-computer scientist can read the book as "the story of grammars and parsing"; he or she can skip the detailed explanations of the algorithms: each algorithm is first explained in general terms. The computer scientist will find much technical detail on a wide array of algorithms. To the expert we offer a systematic bibliography of over

1700 entries. The printed book holds only those entries referenced in the book itself; the full list is available on the web site of this book. All entries in the printed book and about two-thirds of the entries in the web site list come with an annotation; this annotation, or summary, is unrelated to the abstract in the referred article, but rather provides a short explanation of the contents and enough material for the reader to decide if the referred article is worth reading.

No ready-to-run algorithms are given, except for the general context-free parser of Section 17.3. The formulation of a parsing algorithm with sufficient precision to enable a programmer to implement and run it without problems requires a considerable support mechanism that would be out of place in this book and in our experience does little to increase one's understanding of the process involved. The popular methods are given in algorithmic form in most books on compiler construction. The less widely used methods are almost always described in detail in the original publication, for which see Chapter 18.

1.3 Outline of the Contents

Since parsing is concerned with sentences and grammars and since grammars are themselves fairly complicated objects, ample attention is paid to them in Chapter 2. Chapter 3 discusses the principles behind parsing and gives a classification of parsing methods. In summary, parsing methods can be classified as top-down or bottom-up and as directional or non-directional; the directional methods can be further distinguished into deterministic and non-deterministic ones. This situation dictates the contents of the next few chapters.

In Chapter 4 we treat non-directional methods, including Unger and CYK. Chapter 5 forms an intermezzo with the treatment of finite-state automata, which are needed in the subsequent chapters. Chapters 6 through 10 are concerned with directional methods, as follows. Chapter 6 covers non-deterministic directional top-down parsers (recursive descent, Definite Clause Grammars), Chapter 7 non-deterministic directional bottom-up parsers (Earley). Deterministic methods are treated in Chapters 8 (top-down: LL in various forms) and 9 (bottom-up: LR methods). Chapter 10 covers non-canonical parsers, parsers that determine the nodes of a parse tree in a not strictly top-down or bottom-up order (for example left-corner). Non-deterministic versions of the above deterministic methods (for example the GLR parser) are described in Chapter 11.

The next four chapters are concerned with material that does not fit the above framework. Chapter 12 shows a number of recent techniques, both deterministic and non-deterministic, for parsing substrings of complete sentences in a language. Another recent development, in which parsing is viewed as intersecting a context-free grammar with a finite-state automaton is covered in Chapter 13. A few of the numerous parallel parsing algorithms are explained in Chapter 14, and a few of the numerous proposals for non-Chomsky language formalisms are explained in Chapter 15, with their parsers. That completes the parsing methods per se.

Error handling for a selected number of methods is treated in Chapter 16, and Chapter 17 discusses practical parser writing and use.

1.4 The Annotated Bibliography

The annotated bibliography is presented in Chapter 18 both in the printed book and, in a much larger version, on the web site of this book. It is an easily accessible and essential supplement of the main body of the book. Rather than listing all publications in author-alphabetic order, the bibliography is divided into a number of named sections, each concerned with a particular aspect of parsing; there are 25 of them in the printed book and 30 in the web bibliography. Within the sections, the publications are listed chronologically. An author index at the end of the book replaces the usual alphabetic list of publications. A numerical reference placed in brackets is used in the text to refer to a publication. For example, the annotated reference to Earley's publication of the Earley parser is indicated in the text by [14] and can be found on page 578, in the entry marked 14.

2

Grammars as a Generating Device

2.1 Languages as Infinite Sets

In computer science as in everyday parlance, a "grammar" serves to "describe" a "language". If taken at face value, this correspondence, however, is misleading, since the computer scientist and the naive speaker mean slightly different things by the three terms. To establish our terminology and to demarcate the universe of discourse, we shall examine the above terms, starting with the last one.

2.1.1 Language

To the larger part of mankind, language is first and foremost a means of communication, to be used almost unconsciously, certainly so in the heat of a debate. Communication is brought about by sending messages, through air vibrations or through written symbols. Upon a closer look the language messages ("utterances") fall apart into sentences, which are composed of words, which in turn consist of symbol sequences when written. Languages can differ on all three levels of composition. The script can be slightly different, as between English and Irish, or very different, as between English and Chinese. Words tend to differ greatly, and even in closely related languages people call *un cheval* or *ein Pferd*, that which is known to others as *a horse*. Differences in sentence structure are often underestimated; even the closely related Dutch often has an almost Shakespearean word order: "*Ik geloof je niet*", "*I believe you not*", and more distantly related languages readily come up with constructions like the Hungarian "*Pénzem van*", "*Money-my is*", where the English say "*I have money*".

The computer scientist takes a very abstracted view of all this. Yes, a language has sentences, and these sentences possess structure; whether they communicate something or not is not his concern, but information may possibly be derived from their structure and then it is quite all right to call that information the "meaning" of the sentence. And yes, sentences consist of words, which he calls "tokens", each possibly carrying a piece of information, which is its contribution to the meaning of

the whole sentence. But no, words cannot be broken down any further. This does not worry the computer scientist. With his love of telescoping solutions and multi-level techniques, he blithely claims that if words turn out to have structure after all, they are sentences in a different language, of which the letters are the tokens.

The practitioner of formal linguistics, henceforth called the formal-linguist (to distinguish him from the "formal linguist", the specification of whom is left to the imagination of the reader) again takes an abstracted view of this. A language is a "set" of sentences, and each sentence is a "sequence" of "symbols"; that is all there is: no meaning, no structure, either a sentence belongs to the language or it does not. The only property of a symbol is that it has an identity; in any language there are a certain number of different symbols, the *alphabet*, and that number must be finite. Just for convenience we write these symbols as a, b, c, ..., but ✆, ✈, ❒, ... would do equally well, as long as there are enough symbols. The word *sequence* means that the symbols in each sentence are in a fixed order and we should not shuffle them. The word *set* means an unordered collection with all the duplicates removed. A set can be written down by writing the objects in it, surrounded by curly brackets. All this means that to the formal-linguist the following is a language: *a, b, ab, ba*, and so is {*a, aa, aaa, aaaa,* ... } although the latter has notational problems that will be solved later. In accordance with the correspondence that the computer scientist sees between sentence/word and word/letter, the formal-linguist also calls a sentence a *word* and he says that "the word *ab* is in the language {*a, b, ab, ba*}".

Now let us consider the implications of these compact but powerful ideas.

To the computer scientist, a language is a probably infinitely large set of sentences, each composed of tokens in such a way that it has structure; the tokens and the structure cooperate to describe the semantics of the sentence, its "meaning" if you will. Both the structure and the semantics are new, that is, were not present in the formal model, and it is his responsibility to provide and manipulate them both. To a computer scientist $3+4\times5$ is a sentence in the language of "arithmetics on single digits" ("single digits" to avoid having an infinite number of symbols); its structure can be shown by inserting parentheses: $(3+(4\times5))$; and its semantics is probably 23.

To the linguist, whose view of languages, it has to be conceded, is much more normal than that of either of the above, a language is an infinite set of possibly interrelated sentences. Each sentence consists, in a structured fashion, of words which have a meaning in the real world. Structure and words together give the sentence a meaning, which it communicates. Words, again, possess structure and are composed of letters; the letters cooperate with some of the structure to give a meaning to the word. The heavy emphasis on semantics, the relation with the real world and the integration of the two levels sentence/word and word/letters are the domain of the linguist. "*The circle spins furiously*" is a sentence, "*The circle sleeps red*" is nonsense.

The formal-linguist holds his views of language because he wants to study the fundamental properties of languages in their naked beauty; the computer scientist holds his because he wants a clear, well-understood and unambiguous means of describing objects in the computer and of communication with the computer, a most

exacting communication partner, quite unlike a human; and the linguist holds his view of language because it gives him a formal tight grip on a seemingly chaotic and perhaps infinitely complex object: natural language.

2.1.2 Grammars

Everyone who has studied a foreign language knows that a grammar is a book of rules and examples which describes and teaches the language. Good grammars make a careful distinction between the sentence/word level, which they often call *syntax* or *syntaxis* and the word/letter level, which they call *morphology*. Syntax contains rules like "*pour que* is followed by the subjunctive, but *parce que* is not". Morphology contains rules like "the plural of an English noun is formed by appending an *-s*, except when the word ends in *-s*, *-sh*, *-o*, *-ch* or *-x*, in which case *-es* is appended, or when the word has an irregular plural."

We skip the computer scientist's view of a grammar for the moment and proceed immediately to that of the formal-linguist. His view is at the same time very abstract and quite similar to the layman's: a grammar is any exact, finite-size, complete description of the language, i.e., of the set of sentences. This is in fact the school grammar, with the fuzziness removed. Although it will be clear that this definition has full generality, it turns out that it is too general, and therefore relatively powerless. It includes descriptions like "the set of sentences that could have been written by Chaucer"; platonically speaking this defines a set, but we have no way of creating this set or testing whether a given sentence belongs to this language. This particular example, with its "could have been" does not worry the formal-linguist, but there are examples closer to his home that do. "The longest block of consecutive sevens in the decimal expansion of π" describes a language that has at most one word in it (and then that word will consist of sevens only), and as a definition it is exact, of finite-size and complete. One bad thing with it, however, is that one cannot find this word: suppose one finds a block of one hundred sevens after billions and billions of digits, there is always a chance that further on there is an even longer block. And another bad thing is that one cannot even know if this longest block exists at all. It is quite possible that, as one proceeds further and further up the decimal expansion of π, one would find longer and longer stretches of sevens, probably separated by ever-increasing gaps. A comprehensive theory of the decimal expansion of π might answer these questions, but no such theory exists.

For these and other reasons, the formal-linguists have abandoned their static, platonic view of a grammar for a more constructive one, that of the generative grammar: a *generative grammar* is an exact, fixed-size recipe for constructing the sentences in the language. This means that, following the recipe, it must be possible to construct each sentence of the language (in a finite number of actions) and no others. This does not mean that, given a sentence, the recipe tells us *how* to construct that particular sentence, only that it is possible to do so. Such recipes can have several forms, of which some are more convenient than others.

The computer scientist essentially subscribes to the same view, often with the additional requirement that the recipe should imply how a sentence can be constructed.

2.1.3 Problems with Infinite Sets

The above definition of a language as a possibly infinite set of sequences of symbols and of a grammar as a finite recipe to generate these sentences immediately gives rise to two embarrassing questions:

1. How can finite recipes generate enough infinite sets of sentences?
2. If a sentence is just a sequence and has no structure and if the meaning of a sentence derives, among other things, from its structure, how can we assess the meaning of a sentence?

These questions have long and complicated answers, but they do have answers. We shall first pay some attention to the first question and then devote the main body of this book to the second.

2.1.3.1 Infinite Sets from Finite Descriptions

In fact there is nothing wrong with getting a single infinite set from a single finite description: "the set of all positive integers" is a very finite-size description of a definitely infinite-size set. Still, there is something disquieting about the idea, so we shall rephrase our question: "Can all languages be described by finite descriptions?" As the lead-up already suggests, the answer is "No", but the proof is far from trivial. It is, however, very interesting and famous, and it would be a shame not to present at least an outline of it here.

2.1.3.2 Descriptions can be Enumerated

The proof is based on two observations and a trick. The first observation is that descriptions can be listed and given a number. This is done as follows. First, take all descriptions of size one, that is, those of only one letter long, and sort them alphabetically. This is the beginning of our list. Depending on what, exactly, we accept as a description, there may be zero descriptions of size one, or 27 (all letters + space), or 95 (all printable ASCII characters) or something similar; this is immaterial to the discussion which follows.

Second, we take all descriptions of size two, sort them alphabetically to give the second chunk on the list, and so on for lengths 3, 4 and further. This assigns a position on the list to each and every description. Our description "the set of all positive integers", for example, is of size 32, not counting the quotation marks. To find its position on the list, we have to calculate how many descriptions there are with less than 32 characters, say L. We then have to generate all descriptions of size 32, sort them and determine the position of our description in it, say P, and add the two numbers L and P. This will, of course, give a huge number[1] but it does ensure that the description is on the list, in a well-defined position; see Figure 2.1.

[1] Some calculations tell us that, under the ASCII-128 assumption, the number is 248 17168 89636 37891 49073 14874 06454 89259 38844 52556 26245 57755 89193 30291, or roughly 2.5×10^{67}.

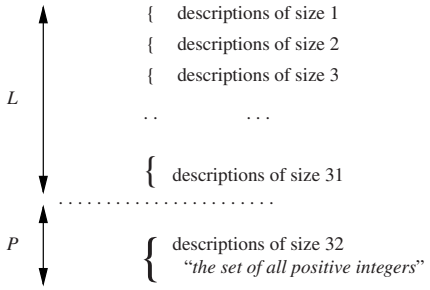

L
P
{ descriptions of size 1
{ descriptions of size 2
{ descriptions of size 3
. . . . . .
{ descriptions of size 31
. . . . . .
{ descriptions of size 32
"the set of all positive integers"

Fig. 2.1. List of all descriptions of length 32 or less

Two things should be pointed out here. The first is that just listing all descriptions alphabetically, without reference to their lengths, would not do: there are already infinitely many descriptions starting with an "a" and no description starting with a higher letter could get a number on the list. The second is that there is no need to actually do all this. It is just a thought experiment that allows us to examine and draw conclusions about the behavior of a system in a situation which we cannot possibly examine physically.

Also, there will be many nonsensical descriptions on the list; it will turn out that this is immaterial to the argument. The important thing is that all meaningful descriptions are on the list, and the above argument ensures that.

2.1.3.3 Languages are Infinite Bit-Strings

We know that words (sentences) in a language are composed of a finite set of symbols; this set is called quite reasonably the "alphabet". We will assume that the symbols in the alphabet are ordered. Then the words in the language can be ordered too. We shall indicate the alphabet by Σ.

Now the simplest language that uses alphabet Σ is that which consists of all words that can be made by combining letters from the alphabet. For the alphabet $\Sigma = \{a, b\}$ we get the language { , *a, b, aa, ab, ba, bb, aaa,* ... }. We shall call this language Σ^*, for reasons to be explained later; for the moment it is just a name.

The set notation Σ^* above started with " { , *a,*", a remarkable construction; the first word in the language is the *empty word*, the word consisting of zero *a*s and zero *b*s. There is no reason to exclude it, but, if written down, it may easily be overlooked, so we shall write it as ε (epsilon), regardless of the alphabet. So, $\Sigma^* = \{ \varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots \}$. In some natural languages, forms of the present tense of the verb "to be" are empty words, giving rise to sentences of the form "I student", meaning "I am a student." Russian and Hebrew are examples of this.

Since the symbols in the alphabet Σ are ordered, we can list the words in the language Σ^*, using the same technique as in the previous section: First, all words of size zero, sorted; then all words of size one, sorted; and so on. This is actually the order already used in our set notation for Σ^*.

The language Σ^* has the interesting property that all languages using alphabet Σ are subsets of it. That means that, given another possibly less trivial language over Σ, called L, we can go through the list of words in Σ^* and put ticks on all words that are in L. This will cover all words in L, since Σ^* contains any possible word over Σ.

Suppose our language L is "the set of all words that contain more *a*s than *b*s". L is the set {*a, aa, aab, aba, baa,* ...}. The beginning of our list, with ticks, will look as follows:

	ε
✔	*a*
	b
✔	*aa*
	ab
	ba
	bb
✔	*aaa*
✔	*aab*
✔	*aba*
	abb
✔	*baa*
	bab
	bba
	bbb
✔	*aaaa*
...	...

Given the alphabet with its ordering, the list of blanks and ticks alone is entirely sufficient to identify and describe the language. For convenience we write the blank as a 0 and the tick as a 1 as if they were bits in a computer, and we can now write $L = 0101000111010001\cdots$ (and $\Sigma^* = 1111111111111111\cdots$). It should be noted that this is true for *any* language, be it a formal language like L, a programming language like Java or a natural language like English. In English, the 1s in the bit-string will be very scarce, since hardly any arbitrary sequence of words is a good English sentence (and hardly any arbitrary sequence of letters is a good English word, depending on whether we address the sentence/word level or the word/letter level).

2.1.3.4 Diagonalization

The previous section attaches the infinite bit-string $0101000111010001\cdots$ to the description "the set of all the words that contain more *a*s than *b*s". In the same vein we can attach such bit-strings to all descriptions. Some descriptions may not yield a language, in which case we can attach an arbitrary infinite bit-string to it. Since all descriptions can be put on a single numbered list, we get, for example, the following picture:

Description	Language
Description #1	$000000100\cdots$
Description #2	$110010001\cdots$
Description #3	$011011010\cdots$
Description #4	$110011010\cdots$
Description #5	$100000011\cdots$
Description #6	$111011011\cdots$
...	...

At the left we have all descriptions, at the right all languages they describe. We now claim that many languages exist that are not on the list of languages above: the above list is far from complete, although the list of descriptions is complete. We shall prove this by using the diagonalization process ("Diagonalverfahren") of Cantor.

Consider the language $C = 100110\cdots$, which has the property that its n-th bit is unequal to the n-th bit of the language described by Description #n. The first bit of C is a 1, because the first bit for Description #1 is a 0; the second bit of C is a 0, because the second bit for Description #2 is a 1, and so on. C is made by walking the NW to SE diagonal of the language field and copying the opposites of the bits we meet. This is the diagonal in Figure 2.2(a). The language C cannot be on the list! It

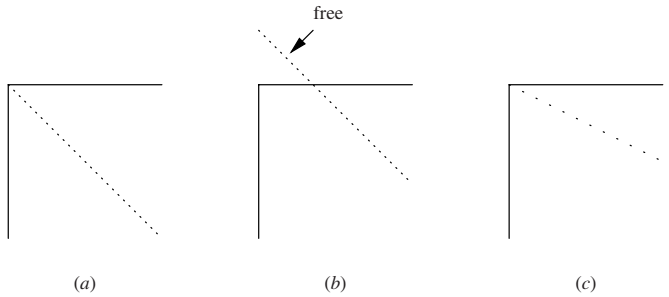


Fig. 2.2. "Diagonal" languages along n (a), $n+10$ (b), and $2n$ (c)

cannot be on line 1, since its first bit differs (is made to differ, one should say) from that on line 1, and in general it cannot be on line n, since its n-th bit will differ from that on line n, by definition.

So, in spite of the fact that we have exhaustively listed all possible finite descriptions, we have at least one language that has no description on the list. But there exist more languages that are not on the list. Construct, for example, the language whose $n+10$-th bit differs from the $n+10$-th bit in Description #n. Again it cannot be on the list since for every $n > 0$ it differs from line n in the $n+10$-th bit. But that means that bits $1\ldots9$ play no role, and can be chosen arbitrarily, as shown in Figure 2.2(b); this yields another $2^9 = 512$ languages that are not on the list. And we can do even much better than that! Suppose we construct a language whose $2n$-th bit differs from the $2n$-th bit in Description #n (c). Again it is clear that it cannot be on the list, but now every odd bit is left unspecified and can be chosen freely! This allows us to create

freely an infinite number of languages none of which allows a finite description; see the slanting diagonal in Figure 2.2. In short, for every language that can be described there are infinitely many that cannot.

The diagonalization technique is described more formally in most books on theoretical computer science; see e.g., Rayward-Smith [393, pp. 5-6], or Sudkamp [397, Section 1.4].

2.1.3.5 Discussion

The above demonstration shows us several things. First, it shows the power of treating languages as formal objects. Although the above outline clearly needs considerable amplification and substantiation to qualify as a proof (for one thing it still has to be clarified why the above explanation, which defines the language *C*, is not itself on the list of descriptions; see Problem 2.1, it allows us to obtain insight into properties not otherwise assessable.

Secondly, it shows that we can only describe a tiny subset (not even a fraction) of all possible languages: there is an infinity of languages out there, forever beyond our reach.

Thirdly, we have proved that, although there are infinitely many descriptions and infinitely many languages, these infinities are not equal to each other, and the latter is larger than the former. These infinities are called $\aleph_0$ and $\aleph_1$ by Cantor, and the above is just a special case of his proof that $\aleph_0 < \aleph_1$.

2.1.4 Describing a Language through a Finite Recipe

A good way to build a set of objects is to start with a small object and to give rules for how to add to it and construct new objects from it. "Two is an even number and the sum of two even numbers is again an even number" effectively generates the set of all even numbers. Formalists will add "and no other numbers are even", but we will take that as understood.

Suppose we want to generate the set of all enumerations of names, of the type "Tom, Dick and Harry", in which all names but the last two are separated by commas. We will not accept "Tom, Dick, Harry" nor "Tom and Dick and Harry", but we shall not object to duplicates: "Grubb, Grubb and Burrowes"[2] is all right. Although these are not complete sentences in normal English, we shall still call them "sentences" since that is what they are in our midget language of name enumerations. A simple-minded recipe would be:

0. Tom is a name, Dick is a name, Harry is a name;
1. a name is a sentence;
2. a sentence followed by a comma and a name is again a sentence;
3. before finishing, if the sentence ends in ", name", replace it by "and name".

[2] *The Hobbit*, by J.R.R. Tolkien, Allen and Unwin, 1961, p. 311.

Although this will work for a cooperative reader, there are several things wrong with it. Clause 3 is especially wrought with trouble. For example, the sentence does not really end in ", name", it ends in ", Dick" or such, and "name" is just a symbol that stands for a real name; such symbols cannot occur in a real sentence and must in the end be replaced by a real name as given in clause 0. Likewise, the word "sentence" in the recipe is a symbol that stands for all the actual sentences. So there are two kinds of symbols involved here: real symbols, which occur in finished sentences, like "Tom", "Dick", a comma and the word "and"; and there are intermediate symbols, like "sentence" and "name" that cannot occur in a finished sentence. The first kind corresponds to the words or *tokens* explained above; the technical term for them is *terminal symbols* (or *terminals* for short). The intermediate symbols are called *non-terminals*, a singularly uninspired term. To distinguish them, we write terminals in lower case letters and start non-terminals with an upper case letter. Non-terminals are called *(grammar) variables* or *syntactic categories* in linguistic contexts.

To stress the generative character of the recipe, we shall replace "X is a Y" by "Y may be replaced by X": if "tom" is an instance of a Name, then everywhere we have a Name we may narrow it down to "tom". This gives us:

0. Name may be replaced by "tom"
   Name may be replaced by "dick"
   Name may be replaced by "harry"
1. Sentence may be replaced by Name
2. Sentence may be replaced by Sentence, Name
3. ", Name" at the end of a Sentence must be replaced by "and Name" before Name is replaced by any of its replacements
4. a sentence is finished only when it no longer contains non-terminals
5. we start our replacement procedure with Sentence

Clause 0 through 3 describe replacements, but 4 and 5 are different. Clause 4 is not specific to this grammar. It is valid generally and is one of the rules of the game. Clause 5 tells us where to start generating. This name is quite naturally called the *start symbol*, and it is required for every grammar.

Clause 3 still looks worrisome; most rules have "may be replaced", but this one has "must be replaced", and it refers to the "end of a Sentence". The rest of the rules work through replacement, but the problem remains how we can use replacement to test for the end of a Sentence. This can be solved by adding an end marker after it. And if we make the end marker a non-terminal which cannot be used anywhere except in the required replacement from ", Name" to "and Name", we automatically enforce the restriction that no sentence is finished unless the replacement test has taken place. For brevity we write $\rightarrow$ instead of "may be replaced by"; since terminal and non-terminal symbols are now identified as technical objects we shall write them in a typewriter-like typeface. The part before the $\rightarrow$ is called the *left-hand side*, the part after it the *right-hand side*. This results in the recipe in Figure 2.3.

This is a simple and relatively precise form for a recipe, and the rules are equally straightforward: start with the start symbol, and keep replacing until there are no non-terminals left.

```
0.          Name  →  tom
            Name  →  dick
            Name  →  harry
1.     Sentence   →  Name
       Sentence   →  List End
2.          List  →  Name
            List  →  List , Name
3.  , Name End    →  and Name
4.  the start symbol is Sentence
```

Fig. 2.3. A finite recipe for generating strings in the t, d & h language

2.2 Formal Grammars

The above recipe form, based on replacement according to rules, is strong enough to serve as a basis for formal grammars. Similar forms, often called "rewriting systems", have a long history among mathematicians, and were already in use several centuries B.C. in India (see, for example, Bhate and Kak [411]). The specific form of Figure 2.3 was first studied extensively by Chomsky [385]. His analysis has been the foundation for almost all research and progress in formal languages, parsers and a considerable part of compiler construction and linguistics.

2.2.1 The Formalism of Formal Grammars

Since formal languages are a branch of mathematics, work in this field is done in a special notation. To show some of its flavor, we shall give the formal definition of a grammar and then explain why it describes a grammar like the one in Figure 2.3. The formalism used is indispensable for correctness proofs, etc., but not for understanding the principles; it is shown here only to give an impression and, perhaps, to bridge a gap.

Definition 2.1: A *generative grammar* is a 4-tuple (V_N, V_T, R, S) such that

(1) V_N and V_T are finite sets of symbols,
(2) $V_N \cap V_T = \emptyset$,
(3) R is a set of pairs (P, Q) such that
  (3a) $P \in (V_N \cup V_T)^+$ and
  (3b) $Q \in (V_N \cup V_T)^*$,
(4) $S \in V_N$

A *4-tuple* is just an object consisting of 4 identifiable parts; they are the nonterminals, the terminals, the rules and the start symbol, in that order. The above definition does not tell this, so this is for the teacher to explain. The set of nonterminals is named V_N and the set of terminals V_T. For our grammar we have:

V_N = {**Name**, **Sentence**, **List**, **End**}
V_T = {**tom**, **dick**, **harry**, **,**, **and**}

(note the , in the set of terminal symbols).

The intersection of V_N and V_T (2) must be empty, indicated by the symbol for the empty set, $\emptyset$. So the non-terminals and the terminals may not have a symbol in common, which is understandable.

R is the set of all rules (3), and P and Q are the left-hand sides and right-hand sides, respectively. Each P must consist of sequences of one or more non-terminals and terminals and each Q must consist of sequences of zero or more non-terminals and terminals. For our grammar we have:

R = {(**Name**, **tom**), (**Name**, **dick**), (**Name**, **harry**),
(**Sentence**, **Name**), (**Sentence**, **List End**), (**List**, **Name**),
(**List**, **List , Name**), (**, Name End**, **and Name**)}

Note again the two different commas.

The start symbol S must be an element of V_N, that is, it must be a non-terminal:

S = **Sentence**

This concludes our field trip into formal linguistics. In short, the mathematics of formal languages is a language, a language that has to be learned; it allows very concise expression of *what* and *how* but gives very little information on *why*. Consider this book a translation and an exegesis.

2.2.2 Generating Sentences from a Formal Grammar

The grammar in Figure 2.3 is what is known as a *phrase structure grammar* for our **t,d&h** language (often abbreviated to *PS grammar*). There is a more compact notation, in which several right-hand sides for one and the same left-hand side are grouped together and then separated by vertical bars, |. This bar belongs to the formalism, just as the arrow →, and can be read "or else". The right-hand sides separated by vertical bars are also called *alternatives*. In this more concise form our grammar becomes

```
0.          Name  →  tom | dick | harry
1.   Sentence_s  →  Name | List End
2.          List  →  Name | Name , List
3.  , Name End    →  and Name
```

where the non-terminal with the subscript $_s$ is the start symbol. (The subscript identifies the symbol, not the rule.)

Now let us generate our initial example from this grammar, using replacement according to the above rules only. We obtain the following successive forms for **Sentence**:

Intermediate form	Rule used	Explanation
`Sentence`		the start symbol
`List End`	`Sentence → List End`	rule 1
`Name , List End`	`List → Name , List`	rule 2
`Name , Name , List End`	`List → Name , List`	rule 2
`Name , Name , Name End`	`List → Name`	rule 2
`Name , Name and Name`	`, Name End → and Name`	rule 3
`tom , dick and harry`		rule 0, three times

The intermediate forms are called *sentential forms*. If a sentential form contains no non-terminals it is called a *sentence* and belongs to the generated language. The transitions from one line to the next are called *production steps* and the rules are called *production rules*, for obvious reasons.

The production process can be made more visual by drawing connective lines between corresponding symbols, using a "graph". A *graph* is a set of *nodes* connected by a set of *edges*. A node can be thought of as a point on paper, and an edge as a line, where each line connects two points; one point may be the end point of more than one line. The nodes in a graph are usually "labeled", which means that they have been given names, and it is convenient to draw the nodes on paper as bubbles with their names in them, rather than as points. If the edges are arrows, the graph is a *directed graph*; if they are lines, the graph is *undirected*. Almost all graphs used in parsing techniques are directed.

The graph corresponding to the above production process is shown in Figure 2.4. Such a picture is called a *production graph* or *syntactic graph* and depicts the syntactic structure (with regard to the given grammar) of the final sentence. We see that the production graph normally fans out downwards, but occasionally we may see starlike constructions, which result from rewriting a group of symbols.

A *cycle* in a graph is a path from a node *N* following the arrows, leading back to *N*. A production graph cannot contain cycles; we can see that as follows. To get a cycle we would need a non-terminal node *N* in the production graph that has produced children that are directly or indirectly *N* again. But since the production process always makes new copies for the nodes it produces, it cannot produce an already existing node. So a production graph is always "acyclic"; *d*irected *a*cyclic *g*raphs are called *dags*.

It is patently impossible to have the grammar generate **`tom, dick, harry`**, since any attempt to produce more than one name will drag in an **`End`** and the only way to get rid of it again (and get rid of it we must, since it is a non-terminal) is to have it absorbed by rule 3, which will produce the **`and`**. Amazingly, we have succeeded in implementing the notion "must replace" in a system that only uses "may replace"; looking more closely, we see that we have split "must replace" into "may replace" and "must not be a non-terminal".

Apart from our standard example, the grammar will of course also produce many other sentences; examples are

```
harry and tom
harry
tom, tom, tom and tom
```

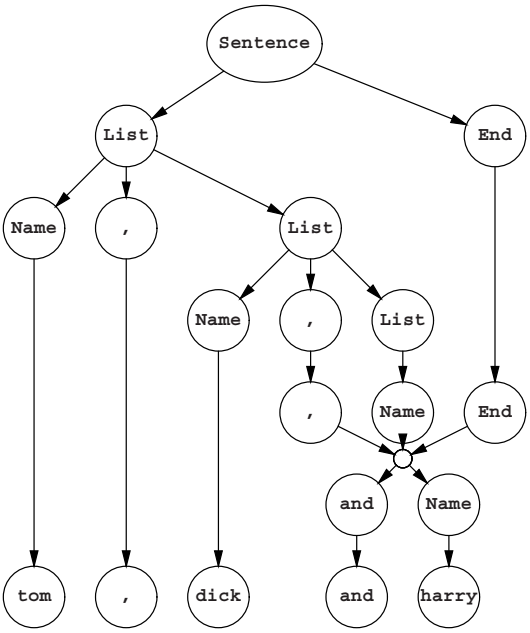


Fig. 2.4. Production graph for a sentence

and an infinity of others. A determined and foolhardy attempt to generate the incorrect form without the **`and`** will lead us to sentential forms like

```
tom, dick, harry End
```

which are not sentences and to which no production rule applies. Such forms are called *blind alleys*. As the right arrow in a production rule already suggests, the rule may not be applied in the reverse direction.

2.2.3 The Expressive Power of Formal Grammars

The main property of a formal grammar is that it has production rules, which may be used for rewriting part of the sentential form (= sentence under construction) and a starting symbol which is the mother of all sentential forms. In the production rules we find non-terminals and terminals; finished sentences contain terminals only. That is about it: the rest is up to the creativity of the grammar writer and the sentence producer.

This is a framework of impressive frugality and the question immediately rises: Is it sufficient? That is hard to say, but if it is not, we do not have anything more expressive. Strange as it may sound, all other methods known to mankind for generating sets have been proved to be equivalent to or less powerful than a phrase structure grammar. One obvious method for generating a set is, of course, to write a program generating it, but it has been proved that any set that can be generated

by a program can be generated by a phrase structure grammar. There are even more arcane methods, but all of them have been proved not to be more expressive. On the other hand there is no proof that no such stronger method can exist. But in view of the fact that many quite different methods all turn out to halt at the same barrier, it is highly unlikely[3] that a stronger method will ever be found. See, e.g. Révész [394, pp 100-102].

As a further example of the expressive power we shall give a grammar for the movements of a Manhattan turtle. A *Manhattan turtle* moves in a plane and can only move north, east, south or west in distances of one block. The grammar of Figure 2.5 produces all paths that return to their own starting point. As to rule 2, it should be

```
1.          Move_S  →  north Move south | east Move west | ε
2.      north east  →  east north
       north south  →  south north
       north west   →  west north
       east north   →  north east
       east south   →  south east
        east west   →  west east
      south north   →  north south
       south east   →  east south
       south west   →  west south
       west north   →  north west
        west east   →  east west
       west south   →  south west
```

Fig. 2.5. Grammar for the movements of a Manhattan turtle

noted that many authors require at least one of the symbols in the left-hand side to be a non-terminal. This restriction can always be enforced by adding new non-terminals.

The simple round trip **`north east south west`** is produced as shown in Figure 2.6 (names abbreviated to their first letter). Note the empty alternative in rule

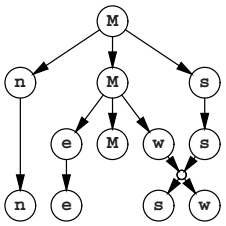


Fig. 2.6. How the grammar of Figure 2.5 produces a round trip

1 (the ε), which results in the dying out of the third **`M`** in the above production graph.

[3] Paul Vitány has pointed out that if scientists call something "highly unlikely" they are still generally not willing to bet a year's salary on it, double or quit.

2.3 The Chomsky Hierarchy of Grammars and Languages

The grammars from Figures 2.3 and 2.5 are easy to understand and indeed some simple phrase structure grammars generate very complicated sets. The grammar for any given set is, however, usually far from simple. (We say "*The* grammar for a given set" although there can be, of course, infinitely many grammars for a set. By *the* grammar for a set, we mean any grammar that does the job and is not obviously overly complicated.) Theory says that if a set can be generated at all (for example, by a program) it can be generated by a phrase structure grammar, but theory does not say that it will be easy to do so, or that the grammar will be understandable. In this context it is illustrative to try to write a grammar for those Manhattan turtle paths in which the turtle is never allowed to the west of its starting point (Problem 2.3).

Apart from the intellectual problems phrase structure grammars pose, they also exhibit fundamental and practical problems. We shall see that no general parsing algorithm for them can exist, and all known special parsing algorithms are either very inefficient or very complex; see Section 3.4.2.

The desire to restrict the unmanageability of phrase structure grammars, while keeping as much of their generative powers as possible, has led to the *Chomsky hierarchy* of grammars. This hierarchy distinguishes four types of grammars, numbered from 0 to 3; it is useful to include a fifth type, called Type 4 here. Type 0 grammars are the (unrestricted) phrase structure grammars of which we have already seen examples. The other types originate from applying more and more restrictions to the allowed form of the rules in the grammar. Each of these restrictions has far-reaching consequences; the resulting grammars are gradually easier to understand and to manipulate, but are also gradually less powerful. Fortunately, these less powerful types are still very useful, actually more useful even than Type 0.

We shall now consider each of the three remaining types in turn, followed by a trivial but useful fourth type.

For an example of a completely different method of generating Type 0 languages see Geffert [395].

2.3.1 Type 1 Grammars

The characteristic property of a Type 0 grammar is that it may contain rules that transform an arbitrary (non-zero) number of symbols into an arbitrary (possibly zero) number of symbols. Example:

```
, N E -> and N
```

in which three symbols are replaced by two. By restricting this freedom, we obtain Type 1 grammars. Strangely enough there are two, intuitively completely different definitions of Type 1 grammars, which can be easily proved to be equivalent.

2.3.1.1 Two Types of Type 1 Grammars

A grammar is *Type 1 monotonic* if it contains no rules in which the left-hand side consists of more symbols than the right-hand side. This forbids, for example, the rule **`, N E -> and N`**.

A grammar is *Type 1 context-sensitive* if all of its rules are context-sensitive. A rule is *context-sensitive* if actually only one (non-terminal) symbol in its left-hand side gets replaced by other symbols, while we find the others back, undamaged and in the same order, in the right-hand side. Example:

```
Name Comma Name End -> Name and Name End
```

which tells that the rule

```
Comma -> and
```

may be applied if the left context is **`Name`** and the right context is **`Name End`**. The contexts themselves are not affected. The replacement must be at least one symbol long. This means that context-sensitive grammars are always monotonic; see Section 2.5.

Here is a monotonic grammar for our **`t,d&h`** example. In writing monotonic grammars one has to be careful never to produce more symbols than will eventually be produced. We avoid the need to delete the end marker by incorporating it into the rightmost name:

```
      Name  ->  tom | dick | harry
Sentence_S  ->  Name | List
      List  ->  EndName | Name , List
, EndName   ->  and Name
```

where **`EndName`** is a single symbol.

And here is a context-sensitive grammar for it.

```
          Name  ->  tom | dick | harry
    Sentence_S  ->  Name | List
          List  ->  EndName
                    | Name Comma List
Comma EndName   ->  and EndName          context is ... EndName
  and EndName   ->  and Name             context is and ...
         Comma  ->  ,
```

Note that we need an extra non-terminal **`Comma`** to produce the terminal **`and`** in the correct context.

Monotonic and context-sensitive grammars are equally powerful: for each language that can be generated by a monotonic grammar a context-sensitive grammar exists that generates the same language, and vice versa. They are less powerful than the Type 0 grammars, that is, there are languages that can be generated by a Type 0 grammar but not by any Type 1. Strangely enough no simple examples of such languages are known. Although the difference between Type 0 and Type 1 is fundamental and is not just a whim of Mr. Chomsky, grammars for which the difference

matters are too complicated to write down; only their existence can be proved (see e.g., Hopcroft and Ullman [391, pp. 183-184], or Révész [394, p. 98]).

Of course any Type 1 grammar is also a Type 0 grammar, since the class of Type 1 grammars is obtained from the class of Type 0 grammars by applying restrictions. But it would be confusing to call a Type 1 grammar a Type 0 grammar; it would be like calling a cat a mammal: correct but imprecise. A grammar is named after the smallest class (that is, the highest type number) in which it will still fit.

We saw that our **t,d&h** language, which was first generated by a Type 0 grammar, could also be generated by a Type 1 grammar. We shall see that there is also a Type 2 and a Type 3 grammar for it, but no Type 4 grammar. We therefore say that the **t,d&h** language is a Type 3 language, after the most restricted (and simple and amenable) grammar for it. Some corollaries of this are: A Type n language can be generated by a Type n grammar or anything stronger, but not by a weaker Type $n+1$ grammar; and: If a language is generated by a Type n grammar, that does not necessarily mean that there is no (weaker) Type $n+1$ grammar for it. The use of a Type 0 grammar for our **t,d&h** language was a serious case of overkill, just for demonstration purposes.

2.3.1.2 Constructing a Type 1 Grammar

The standard example of a Type 1 language is the set of words that consist of equal numbers of **a**s, **b**s and **c**s, in that order:

$$\underbrace{a\,a\,\ldots\,a}_{n\text{ of them}}\quad\underbrace{b\,b\,\ldots\,b}_{n\text{ of them}}\quad\underbrace{c\,c\,\ldots\,c}_{n\text{ of them}}$$

For the sake of completeness and to show how one writes a Type 1 grammar if one is clever enough, we shall now derive a grammar for this toy language. Starting with the simplest case, we have the rule

0. **S → abc**

Having obtained one instance of **S**, we may want to prepend more **a**s to the beginning; if we want to remember how many there were, we shall have to append something to the end as well at the same time, and that cannot be a **b** or a **c**. We shall use a yet unknown symbol **Q**. The following rule both prepends and appends:

1. **S → aSQ**

If we apply this rule, for example, three times, we get the sentential form

aaabcQQQ

Now, to get **aaabbbccc** from this, each **Q** must be worth one **b** and one **c**, as expected, but we cannot just write

Q → bc

because that would allow **b**s after the first **c**. The above rule would, however, be all right if it were allowed to do replacement only between a **b** on the left and a **c** on the right. There the newly inserted **bc** will do no harm:

2. **bQc → bbcc**

Still, we cannot apply this rule since normally the **Q**s are to the right of the **c**. This can be remedied by allowing a **Q** to hop left over a **c**:

3. **cQ → Qc**

We can now finish our derivation:

aaabcQQ	(3 times rule 1)
aaabQcQ	(rule 3)
aaabbccQ	(rule 2)
aaabbcQc	(rule 3)
aaabbQcc	(rule 3)
aaabbbccc	(rule 2)

It should be noted that the above derivation only shows that the grammar will produce the right strings, and the reader will still have to convince himself that it will not generate other and incorrect strings (Problem 2.4).

The grammar is summarized in Figure 2.7. Since a derivation graph of $\mathbf{a}^3\mathbf{b}^3\mathbf{c}^3$

1. $\mathbf{S}_S \rightarrow \mathbf{abc} \mid \mathbf{aSQ}$
2. $\mathbf{bQc} \rightarrow \mathbf{bbcc}$
3. $\mathbf{cQ} \rightarrow \mathbf{Qc}$

Fig. 2.7. Monotonic grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

is already rather unwieldy, a derivation graph for $\mathbf{a}^2\mathbf{b}^2\mathbf{c}^2$ is given in Figure 2.8. The

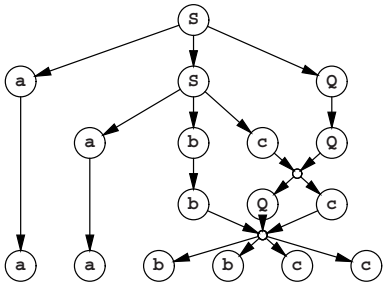


Fig. 2.8. Derivation of **aabbcc**

grammar is monotonic and therefore of Type 1. It can be proved that there is no Type 2 grammar for the language; see Section 2.7.1.

Although only context-sensitive Type 1 grammars can by rights be called context-sensitive grammars (CS grammars), that name is often used even if the grammar is actually monotonic Type 1. There are no standard initials for monotonic, but MT will do.

2.3.2 Type 2 Grammars

Type 2 grammars are called *context-free grammars* (CF grammars) and their relation to context-sensitive grammars is as direct as the name suggests. A context-free grammar is like a context-sensitive grammar, except that both the left and the right contexts are required to be absent (empty). As a result, the grammar may contain only rules that have a single non-terminal on their left-hand side. Sample grammar:

```
0.        Name  -->  tom | dick | harry
1.  Sentence_S  -->  Name | List and Name
2.        List  -->  Name , List | Name
```

2.3.2.1 Production Independence

Since there is always only one symbol on the left-hand side, each node in a production graph has the property that whatever it produces is independent of what its neighbors produce: the productive life of a non-terminal is independent of its context. Starlike forms as we saw in Figures 2.4, 2.6, and 2.8 cannot occur in a context-free production graph, which consequently has a pure *tree*-form and is called a *production tree*. An example is shown in Figure 2.9.

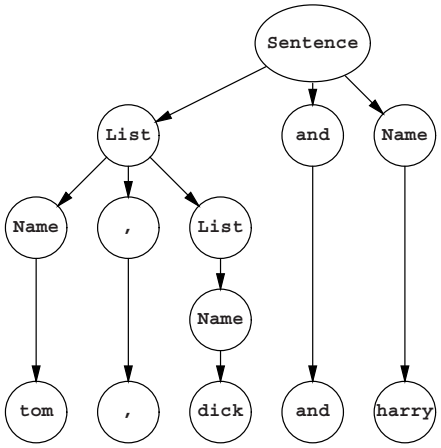


Fig. 2.9. Production tree for a context-free grammar

Since there is only one symbol on the left-hand side, all right-hand sides for a given non-terminal can always be collected in one grammar rule (we have already

done that in the above grammar) and then each grammar rule reads like a definition of the left-hand side:

- A **Sentence** is either a **Name** or a **List** followed by **and** followed by a **Name**.
- A **List** is either a **Name** followed by a **,** followed by a **List**, or it is a **Name**.

This shows that context-free grammars build the strings they produce by two processes: *concatenation* ("... followed by ...") and *choice* ("either ... or ..."). In addition to these processes there is the *identification mechanism* which links the name of a non-terminal used in a right-hand side to its defining rule ("... is a ...").

At the beginning of this chapter we identified a language as a set of strings, the set of terminal productions of the start symbol. The independent production property allows us to extend this definition to any non-terminal in the grammar: each non-terminal produces a set, a language, independent of the other non-terminals. If we write the set of strings produced by A as $\mathcal{L}(A)$ and A has a production rule with, say, two alternatives, $A \rightarrow \alpha|\beta$, then $\mathcal{L}(A) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$, where $\cup$ is the union operator on sets. This corresponds to the choice in the previous paragraph. If α then consists of, say, three members PqR, we have $\mathcal{L}(\alpha) = \mathcal{L}(P) \circ \mathcal{L}(q) \circ \mathcal{L}(R)$, where $\circ$ is the concatenation operator on strings (actually on the strings in the sets). This corresponds to the concatenation above. And $\mathcal{L}(a)$ where a is a terminal is of course the set $\{a\}$. A non-terminal whose language contains ε is called *nullable*. One also says that it "produces empty".

Note that we cannot define a language $\mathcal{L}(Q)$ for the Q in Figure 2.7: Q does not produce anything meaningful by itself. Defining a language for a non-start symbol is possible only for Type 2 grammars and lower, and so is defining a non-start non-terminal as nullable.

Related to the independent production property is the notion of *recursion*. A non-terminal A is recursive if an A in a sentential form can produce something that again contains an A. The production of Figure 2.9 starts with the sentential form **Sentence**, which uses rule 1.2 to produce **List and Name**. The next step could very well be the replacement of the **List** by **Name,List**, using rule 2.1. We see that **List** produces something that again contains **List**:

```
Sentence → List and Name → Name , List and Name
```

List is recursive, more in particular, it is *directly recursive*. The non-terminal **A** in **A→Bc**, **B→dA** is *indirectly recursive*, but not much significance is to be attached to the difference.

It is more important that **List** is right-recursive: a non-terminal A is *right-recursive* if it can produce something that has an A at the right end, as **List** can:

```
List → Name , List
```

Likewise, a non-terminal A is *left-recursive* if it can produce something that has an A at the left end: we could have defined

```
List → List , Name
```

A non-terminal *A* is *self-embedding* if there is a derivation in which *A* produces *A* with something, say α, before it *and* something, say β, after it. Self-embedding describes nesting: α is the part produced when entering another level of nesting; β is the part produced when leaving that level. The best-known example of nesting is the use of parentheses in arithmetic expressions:

```
arith_expression_S  ->  ··· | simple_expression
simple_expression   ->  number | '(' arith_expression ')'
```

A non-terminal can be left-recursive and right-recursive at the same time; it is then self-embedding. **A→Ab | cA | d** is an example.

If no non-terminal in a grammar is recursive, each production step uses up one non-terminal, since that non-terminal will never occur again in that segment. So the production process cannot continue unlimitedly, and a finite language results. Recursion is essential for life in grammars.

2.3.2.2 Some Examples

In the actual world, many things are defined in terms of other things. Context-free grammars are a very concise way to formulate such interrelationships. An almost trivial example is the composition of a book, as given in Figure 2.10. Of course this

```
          Book_S  ->  Preface ChapterSequence Conclusion
         Preface  ->  "PREFACE" ParagraphSequence
 ChapterSequence  ->  Chapter | Chapter ChapterSequence
         Chapter  ->  "CHAPTER" Number ParagraphSequence
ParagraphSequence ->  Paragraph | Paragraph ParagraphSequence
       Paragraph  ->  SentenceSequence
SentenceSequence  ->  ···
                  ···
      Conclusion  ->  "CONCLUSION" ParagraphSequence
```

Fig. 2.10. A simple (and incomplete!) grammar of a book

is a context-free description of a book, so one can expect it to also generate a lot of good-looking nonsense like

```
PREFACE
qwertyuiop
CHAPTER V
asdfghjkl
zxcvbnm,.
CHAPTER II
qazwsxedcrfvtgb
yhnujmikolp
CONCLUSION
All cats say blert when walking through walls.
```

but at least the result has the right structure. Document preparation and text mark-up systems like SGML, HTML and XML use this approach to express and control the basic structure of documents.

A shorter but less trivial example is the language of all elevator motions that return to the same point (a Manhattan turtle restricted to 5th Avenue would make the same movements)

```
ZeroMotion_s  →  up ZeroMotion down ZeroMotion
              |  down ZeroMotion up ZeroMotion
              |  ε
```

(in which we assume that the elevator shaft is infinitely long; it would be, in Manhattan).

If we ignore enough detail we can also recognize an underlying context-free structure in the sentences of a natural language, for example, English:

```
Sentence_s  →  Subject Verb Object
  Subject   →  NounPhrase
   Object   →  NounPhrase
 NounPhrase →  the QualifiedNoun
QualifiedNoun → Noun | Adjective QualifiedNoun
     Noun   →  castle | caterpillar | cats
  Adjective →  well-read | white | wistful | ···
     Verb   →  admires | bark | criticize | ···
```

which produces sentences like:

```
the well-read cats criticize the wistful caterpillar
```

Since, however, no context is incorporated, it will equally well produce the incorrect

```
the cats admires the white well-read castle
```

For keeping context we could use a phrase structure grammar (for a simpler language):

```
Sentence_s  →  Noun Number Verb
  Number    →  Singular | Plural
Noun Singular → castle Singular | caterpillar Singular | ···
Singular Verb → Singular admires | ···
 Singular   →  ε
Noun Plural →  cats Plural | ···
Plural Verb →  Plural bark | Plural criticize | ···
  Plural    →  ε
```

where the markers **Singular** and **Plural** control the production of actual English words. Still, this grammar allows the cats to bark. . . . For a better way to handle context, see the various sections in Chapter 15, especially Van Wijngaarden grammars (Section 15.2) and attribute and affix grammars (Section 15.3).

The bulk of examples of CF grammars originate from programming languages. Sentences in these languages (that is, programs) have to be processed automatically

(that is, by a compiler) and it was soon recognized (around 1958) that this is much easier if the language has a well-defined formal grammar. The syntaxes of all programming languages in use today are defined through formal grammars.

Some authors (for example Chomsky) and some parsing algorithms, require a CF grammar to be monotonic. The only way a CF rule can be non-monotonic is by having an empty right-hand side. Such a rule is called an ε-*rule* and a grammar that contains no such rules is called ε-*free*.

The requirement of being ε-free is not a real restriction, just a nuisance. Almost any CF grammar can be made ε-free by systematic substitution of the ε-rules; the exception is a grammar in which the start symbol already produces ε. The transformation process is explained in detail in Section 4.2.3.1), but it shares with many other grammar transformations the disadvantage that it usually ruins the structure of the grammar. The issue will be discussed further in Section 2.5.

2.3.2.3 Notation Styles

There are several different styles of notation for CF grammars for programming languages, each with endless variants; they are all functionally equivalent. We shall show two main styles here. The first is the *Backus-Naur Form* (*BNF*) which was first used to define ALGOL 60. Here is a sample:

```
<name>::=       tom | dick | harry
<sentence>s::=  <name> | <list> and <name>
<list>::=       <name>, <list> | <name>
```

This form's main properties are the use of angle brackets to enclose non-terminals and of **::=** for "may produce". In some variants, the rules are terminated by a semicolon.

The second style is that of the CF van Wijngaarden grammars. Again a sample:

```
name:       tom symbol; dick symbol; harry symbol.
sentences:  name; list, and symbol, name.
list:       name, comma symbol, list; name.
```

The names of terminal symbols end in **...symbol**; their representations are hardware-dependent and are not defined in the grammar. Rules are properly terminated (with a period). Punctuation is used more or less in the traditional way; for example, the comma binds tighter than the semicolon. The punctuation can be read as follows:

- **:** "is defined as a(n)"
- **;** ", or as a(n)"
- **,** "followed by a(n)"
- **.** ", and as nothing else."

The second rule in the above grammar would then read as: "a sentence is defined as a name, or as a list followed by an and-symbol followed by a name, and as nothing else." Although this notation achieves its full power only when applied in the two-level Van Wijngaarden grammars, it also has its merits on its own: it is formal and still quite readable.

2.3.2.4 Extended CF Grammars

CF grammars are often made both more compact and more readable by introducing special short-hands for frequently used constructions. If we return to the Book grammar of Figure 2.10, we see that rules like:

```
SomethingSequence → Something | Something SomethingSequence
```

occur repeatedly. In an *extended context-free grammar* we can write **Something**$^{+}$ meaning "one or more **Something**s" and we do not need to give a rule for **Something**$^{+}$; the rule

```
Something+ → Something | Something Something+
```

is implicit. Likewise we can use **Something*** for "zero or more **Something**s" and **Something**$^{?}$ for "zero or one **Something**" (that is, "optionally a **Something**"). In these examples, the operators $^{+}$, * and $^{?}$ work on the preceding symbol. Their range can be extended by using parentheses: **(Something ;)**$^{?}$ means "optionally a **Something**-followed-by-a-**;**". These facilities are very useful and allow the Book grammar to be written more efficiently (Figure 2.11). Some styles even allow constructions like **Something**$^{+4}$, meaning "one or more **Something**s with a maximum of 4", or **Something**$^{+}$**,** meaning "one or more **Something**s separated by commas"; this seems to be a case of overdoing a good thing. This notation for grammars is called *Extended BNF* (*EBNF*).

```
Books  →  Preface Chapter+ Conclusion
Preface  →  "PREFACE" Paragraph+
Chapter  →  "CHAPTER" Number Paragraph+
Paragraph  →  Sentence+
Sentence  →  ···
          ···
Conclusion  →  "CONCLUSION" Paragraph+
```

Fig. 2.11. A grammar of a book in EBFN notation

The extensions of an EBNF grammar do not increase its expressive powers: all implicit rules can be made explicit and then a normal CF grammar in BNF notation results. Their strength lies in their user-friendliness. The star in the notation X^{*} with the meaning "a sequence of zero or more Xs" is called the *Kleene star*. If X is a set, X^{*} should be read as "a sequence of zero or more elements of X"; it is the same star that we saw in Σ^{*} in Section 2.1.3.3. Forms involving the repetition operators *, $^{+}$ or $^{?}$ and possibly the separators **(** and **)** are called *regular expressions*. EBNFs, which have regular expressions for their right-hand sides, are for that reason sometimes called *regular right part grammars RRP grammars* which is more descriptive than "extended context free", but which is perceived to be a tongue twister by some.

There are two different schools of thought about the structural meaning of a regular right-hand side. One school maintains that a rule like:

```
Book  →  Preface Chapter+ Conclusion
```

is an abbreviation of

```
Book  →  Preface α Conclusion
   α  →  Chapter | Chapter α
```

as shown above. This is the "(right)recursive" interpretation. It has the advantages that it is easy to explain and that the transformation to "normal" CF is simple. Disadvantages are that the transformation entails anonymous rules (identified by α here) and that the lopsided production tree for, for example, a book of four chapters does not correspond to our idea of the structure of the book; see Figure 2.12.

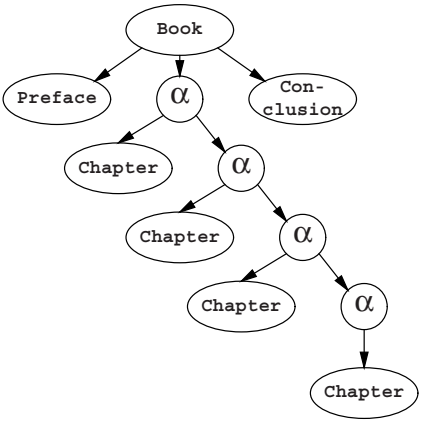


Fig. 2.12. Production tree for the (right)recursive interpretation

The second school claims that

```
Book  →  Preface Chapter+ Conclusion
```

is an abbreviation of

```
Book  →  Preface Chapter Conclusion
      |  Preface Chapter Chapter Conclusion
      |  Preface Chapter Chapter Chapter Conclusion
      |  ···
     ···
```

This is the "iterative" interpretation. It has the advantage that it yields a beautiful production tree (Figure 2.13), but the disadvantages are that it involves an infinite number of production rules and that the nodes in the production tree have a varying fan-out.

Since the implementation of the iterative interpretation is far from trivial, most practical parser generators use the recursive interpretation in some form or another, whereas most research has been done on the iterative interpretation.

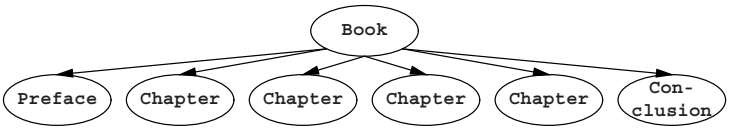


Fig. 2.13. Production tree for the iterative interpretation

2.3.3 Type 3 Grammars

The basic property of CF grammars is that they describe things that nest: an object may contain other objects in various places, which in turn may contain … etc. When during the production process we have finished producing one of the objects, the right-hand side still "remembers" what has to come after it: in the English grammar, after having descended into the depth of the non-terminal **`Subject`** to produce something like **`the wistful cat`**, the right-hand side **`Subject Verb Object`** still remembers that a **`Verb`** must follow. While we are working on the **`Subject`**, the **`Verb`** and **`Object`** remain queued at the right in the sentential form, for example,

```
the wistful QualifiedNoun Verb Object
```

In the right-hand side

```
up ZeroMotion down ZeroMotion
```

after having performed the **`up`** and an arbitrarily complicated **`ZeroMotion`**, the right-hand side still remembers that a **`down`** must follow.

The restriction to Type 3 disallows this recollection of things that came before: a right-hand side may only contain one non-terminal and it must come at the end. This means that there are only two kinds of rules:[4]

- a non-terminal produces zero or more terminals, and
- a non-terminal produces zero or more terminals followed by one non-terminal.

The original Chomsky definition of Type 3 restricts the kinds of rules to

- a non-terminal produces one terminal.
- A non-terminal produces one terminal followed by one non-terminal.

Our definition is equivalent and more convenient, although the conversion to Chomsky Type 3 is not completely trivial.

Type 3 grammars are also called *regular grammars* (RE grammars) or *finite-state grammars* (FS grammars). More precisely the version defined above is called *right-regular* since the only non-terminal in a rule is found at the right end of the right-hand side. This distinguishes them from the *left-regular grammars*, which are subject to the restrictions

[4] There is a natural in-between class, Type 2.5 so to speak, in which only a single non-terminal is allowed in a right-hand side, but where it need not be at the end. This gives us the so-called *linear grammars*.

- a non-terminal produces zero or more terminals
- a non-terminal produces one non-terminal followed by zero or more terminals

where the only non-terminal in a rule is found at the left end of the right-hand side. Left-regular grammars are less intuitive than right-regular ones, occur less frequently, and are more difficult to process, but they do occur occasionally (see for example Section 5.1.1), and need to be considered. They are discussed in Section 5.6.

Given the prevalence of right-regular over left-regular, the term "regular grammar" is usually intended to mean "*right*-regular grammar", and left-regularity is mentioned explicitly. We will follow this convention in this book.

It is interesting to compare the definition of right-regular to that of right-recursive (page 24). A non-terminal *A* is right-*recursive* if it can produce a sentential form that has an *A* at the right end; *A* is right-*regular* if, when it produces a sentential form that contains *A*, the *A* is at the right end.

In analogy to context-free grammars, which are called after what they cannot do, regular grammars could be called "non-nesting grammars".

Since regular grammars are used very often to describe the structure of text on the character level, it is customary for the terminal symbols of a regular grammar to be single characters. We shall therefore write **t** for **Tom**, **d** for **Dick**, **h** for **Harry** and **&** for **and**. Figure 2.14(*a*) shows a right-regular grammar for our **t,d&h** language in this style, 2.14(*b*) a left-regular one.

```
Sentence_S  ->  t | d | h | List
     List   ->  t ListTail | d ListTail | h ListTail
 ListTail   ->  , List | & t | & d | & h
```

(*a*)

```
Sentence_S  ->  t | d | h | List
     List   ->  ListHead & t | ListHead & d | ListHead & h
 ListHead   ->  ListHead , t | ListHead , d | ListHead , h |
                t | d | h
```

(*b*)

Fig. 2.14. Type 3 grammars for the t, d & h language

The production tree for a sentence from a Type 3 (right-regular) grammar degenerates into a "production chain" of non-terminals that drop a sequence of terminals on their left. Figure 2.15 shows an example. Similar chains are formed by left-regular grammars, with terminals dropping to the left.

The deadly repetition exhibited by the grammar of Figure 2.14 is typical of regular grammars and a number of notational devices have been invented to abate this nuisance. The most common one is the use of square brackets to indicate "one out of a set of characters": **[tdh]** is an abbreviation for **t**|**d**|**h**:

```
S_S  ->  [tdh] | L
 L   ->  [tdh] T
 T   ->  , L | & [tdh]
```

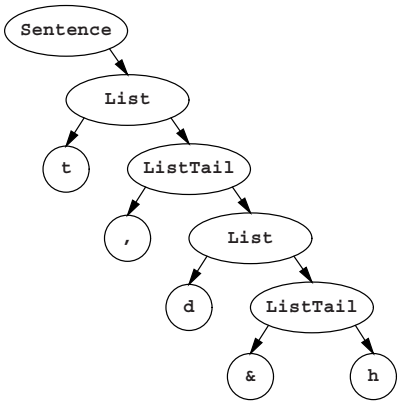


Fig. 2.15. Production chain for a right-regular (Type 3) grammar

which may look more cryptic at first but is actually much more convenient and in fact allows simplification of the grammar to

```
S_S  ->  [tdh] | L
 L   ->  [tdh] , L | [tdh] & [tdh]
```

A second way is to allow macros, names for pieces of the grammar that are substituted properly into the grammar before it is used:

```
Name  ->  t | d | h
 S_S  ->  $Name | L
  L   ->  $Name , L | $Name & $Name
```

The popular parser generator for regular grammars *lex* (Lesk and Schmidt [360]) features both facilities.

If we adhere to the Chomsky definition of Type 3, our grammar will not get smaller than:

```
S_S  ->  t | d | h | t M | d M | h M
 M   ->  , N | & P
 N   ->  t M | d M | h M
 P   ->  t | d | h
```

This form is easier to process but less user-friendly than the *lex* version. We observe here that while the formal-linguist is interested in and helped by minimally sufficient means, the computer scientist values a form in which the concepts underlying the grammar (**`$Name`**, etc.) are easily expressed, at the expense of additional processing.

There are two interesting observations about regular grammars which we want to make here. First, when we use a regular grammar for generating a sentence, the sentential forms will only contain one non-terminal and this will always be at the end; that is where it all happens (using the grammar of Figure 2.14):

```
Sentence_S
List
t ListTail
t , List
t , d ListTail
t , d & h
```

The second observation is that all regular grammars can be reduced considerably in size by using the regular expression operators *, $^+$ and $^?$ introduced in Section 2.3.2 for "zero or more", "one or more" and "optionally one", respectively. Using these operators and `(` and `)` for grouping, we can simplify our grammar to:

```
S_S  ->  (( [tdh], )* [tdh]& )? [tdh]
```

Here the parentheses serve to demarcate the operands of the * and $^?$ operators. Regular expressions exist for all Type 3 grammars. Note that the * and the $^+$ work on what precedes them. To distinguish them from the normal multiplication and addition operators, they are often printed higher than the level text in print, but in computer input they are in line with the rest, and other means must be used to distinguish them.

2.3.4 Type 4 Grammars

The last restriction we shall apply to what is allowed in a production rule is a pretty final one: no non-terminal is allowed in the right-hand side. This removes all the generative power from the mechanism, except for the choosing of alternatives. The start symbol has a (finite) list of alternatives from which we are allowed to choose; this is reflected in the name *finite-choice grammar* (FC grammar).

There is no FC grammar for our `t,d&h` language; if, however, we are willing to restrict ourselves to lists of names of a finite length (say, no more than a hundred), then there is one, since one could enumerate all combinations. For the obvious limit of three names, we get:

```
S_S  ->  [tdh] | [tdh] & [tdh] | [tdh] , [tdh] & [tdh]
```

for a total of $3+3\times3+3\times3\times3 = 39$ production rules.

FC grammars are not part of the official Chomsky hierarchy in that they are not identified by Chomsky. They are nevertheless very useful and are often required as a tail-piece in some process or reasoning. The set of reserved words (*keywords*) in a programming language can be described by an FC grammar. Although not many grammars are FC in their entirety, some of the rules in many grammars are finite-choice. For example, the first rule of our first grammar (Figure 2.3) was FC. Another example of a FC rule was the macro introduced in Section 2.3.3. We do not need the macro mechanism if we change

zero or more terminals

in the definition of a regular grammar to

zero or more terminals or FC non-terminals

In the end, the FC non-terminals will only introduce a finite number of terminals.

2.3.5 Conclusion

The table in Figure 2.16 summarizes the most complicated data structures that can occur in the production of a string, in correlation to the grammar type used. See also Figure 3.15 for the corresponding data types obtained in parsing.

Chomsky type	Grammar type	Most complicated data structure	Example figure
0 / 1	PS / CS	production dag	2.8
2	CF	production tree	2.9
3	FS	production list	2.15
4	FC	production element	—

Fig. 2.16. The most complicated production data structure for the Chomsky grammar types

2.4 Actually Generating Sentences from a Grammar

2.4.1 The Phrase-Structure Case

Until now we have only produced single sentences from our grammars, in an ad hoc fashion, but the purpose of a grammar is to generate all of its sentences. Fortunately there is a systematic way to do so. We shall use the $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ grammar as an example. We start from the start symbol and systematically make all possible substitutions to generate all sentential forms; we just wait and see which ones evolve into sentences and when. Try this by hand for, say, 10 sentential forms. If we are not careful, we are apt to only generate forms like `aSQ`, `aaSQQ`, `aaaSQQQ`, ..., and we will never see a finished sentence. The reason is that we focus too much on a single sentential form: we have to give equal time to all of them. This can be done through the following algorithm, which keeps a queue (that is, a list to which we add at the end and remove from the beginning), of sentential forms.

Start with the start symbol as the only sentential form in the queue. Now continue doing the following:

- Consider the first sentential form in the queue.
- Scan it from left to right, looking for strings of symbols that match the left-hand side of a production rule.
- For each such string found, make enough copies of the sentential form, replace in each one the string that matched a left-hand side of a rule by a different alternative of that rule, and add them all to the end of the queue.
- If the original sentential form does not contain any non-terminals, write it down as a sentence in the language.
- Throw away the original sentential form; it has been fully processed.

If no rule matched, and the sentential form was not a finished sentence, it was a blind alley; they are removed automatically by the above process and leave no trace.

Since the above procedure enumerates all strings in a PS language, PS languages are also called *recursively enumerable* sets, where "recursively" is to be taken to mean "by a possibly recursive algorithm".

The first couple of steps of this process for our $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ grammar from Figure 2.7 are depicted in Figure 2.17. The queue runs to the right, with the first item on

Step	Queue					Result
1	S					
2	abc	aSQ				abc
3	aSQ					
4	aabcQ	aaSQQ				
5	aaSQQ	aabQc				
6	aabQc	aaabcQQ	aaaSQQQ			
7	aaabcQQ	aaaSQQQ	aabbcc			
8	aaaSQQQ	aabbcc	aaabQcQ			
9	aabbcc	aaabQcQ	aaaabcQQQ	aaaaSQQQQ		aabbcc
10	aaabQcQ	aaaabcQQQ	aaaaSQQQQ			
11	aaaabcQQQ	aaaaSQQQQ	aaabbccQ	aaabQQc		
...	...					

Fig. 2.17. The first couple of steps in producing for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

the left. We see that we do not get a sentence for each time we turn the crank; in fact, in this case real sentences will get scarcer and scarcer. The reason is of course that during the process more and more side lines develop, which all require equal attention. Still, we can be certain that every sentence that can be produced, will in the end be produced: we leave no stone unturned. This way of doing things is called *breadth-first production*; computers are better at it than people.

It is tempting to think that it is unnecessary to replace *all* left-hand sides that we found in the top-most sentential form. Why not just replace the first one and wait for the resulting sentential form to come up again and then do the next one? This is wrong, however, since doing the first one may ruin the context for doing the second one. A simple example is the grammar

$$\begin{aligned} \mathtt{S}_S &\rightarrow \mathtt{AC} \\ \mathtt{A} &\rightarrow \mathtt{b} \\ \mathtt{AC} &\rightarrow \mathtt{ac} \end{aligned}$$

First doing $\mathtt{A} \rightarrow \mathtt{b}$ will lead to a blind alley and the grammar will produce nothing. Doing both possible substitutions will lead to the same blind alley, but then there will also be a second sentential form, **ac**. This is also an example of a grammar for which the queue will get empty after a (short) while.

If the grammar is context-free (or regular) there is no context to ruin and it is quite safe to just replace the first (or only) match.

There are two remarks to be made here. First, it is not at all certain that we will indeed obtain a sentence for all our effort: it is quite possible that every new sentential form again contains non-terminals. We should like to know this in advance by examining the grammar, but it can be proven that it is impossible to do so for PS grammars. The formal-linguist says "It is *undecidable* whether a PS grammar produces the empty set", which means that there cannot be an algorithm that will for every PS grammar correctly tell if the grammar produces at least one sentence. This does not mean that we cannot prove for some given grammar that it generates nothing, if that is the case. It means that the proof method used will not work for *all* grammars: we could have a program that correctly says Yes in finite time if the answer is Yes but that takes infinite time if the answer is No. In fact, our generating procedure above is such an algorithm that gives the correct Yes/No answer in infinite time (although we can have an algorithm that gives a Yes/Don't know answer in finite time). Although it is true that because of some deep property of formal languages we cannot always get exactly the answer we want, this does not prevent us from obtaining all kinds of useful information that gets close. We shall see that this is a recurring phenomenon. The computer scientist is aware of but not daunted by the impossibilities from formal linguistics.

The second remark is that when we do get sentences from the above production process, they may be produced in an unexploitable order. For non-monotonic grammars the sentential forms may grow for a while and then suddenly shrink again, perhaps even to the empty string. Formal linguistics proves that there cannot be an algorithm that for all PS grammars produces their sentences in increasing (actually "non-decreasing") length. In other words, the parsing problem for PS grammars is *unsolvable*. (Although the terms are used interchangeably, it seems reasonable to use "undecidable" for yes/no questions and "unsolvable" for problems.)

2.4.2 The CS Case

The above language-generating procedure is also applicable to CS grammars, except for the parts about undecidability. Since the sentential forms under development can never shrink, the strings are produced in monotonic order of increasing length. This means that if the empty string is not the first string, it will never appear and the CS grammar does not produce ε. Also, if we want to know if a given string w is in the language, we can just wait until we see it come up, in which case the answer is Yes, or until we see a longer string come up, in which case the answer is No.

Since the strings in a CS language can be recognized by a possibly recursive algorithm, CS languages are also called *recursive sets*.

2.4.3 The CF Case

When we generate sentences from a CF grammar, many things are a lot simpler. It can still happen that our grammar will never produce a sentence, but now we can test for that beforehand, as follows. First scan the grammar to find all non-terminals that have a right-hand side that contains terminals only or is empty. These non-terminals

are guaranteed to produce something. Now scan again to find non-terminals that have a right-hand side that consists of only terminals and non-terminals that are guaranteed to produce something. This will give us new non-terminals that are guaranteed to produce something. Repeat this until we find no more new such non-terminals. If we have not met the start symbol this way, it will not produce anything.

Furthermore we have seen that if the grammar is CF, we can afford to just rewrite the leftmost non-terminal every time (provided we rewrite it into all its alternatives). Of course we can also consistently rewrite the rightmost non-terminal. Both approaches are similar but different. Using the grammar

```
0.  N   ->  t | d | h
1.  Ss  ->  N | L & N
2.  L   ->  N , L | N
```

let us follow the adventures of the sentential form that will eventually result in **d,h&h**. Although it will go up and down the production queue several times, we only depict here what changes are made to it. Figure 2.18 shows the sentential forms for leftmost and rightmost substitution, with the rules and alternatives involved; for example, (1b) means rule 1 alternative b, the second alternative.

	S		**S**
1b		1b	
	L&N		**L&N**
2a		0c	
	N,L&N		**L&h**
0b		2a	
	d,L&N		**N,L&h**
2b		2b	
	d,N&N		**N,N&h**
0c		0c	
	d,h&N		**N,h&h**
0c		0b	
	d,h&h		**d,h&h**

Fig. 2.18. Sentential forms leading to **d,h&h**, with leftmost and rightmost substitution

The sequences of production rules used are not as similar as we would expect. Of course in grand total the same rules and alternatives are applied, but the sequences are neither equal nor each other's mirror image, nor is there any other obvious relationship. Both sequences define the same production tree (Figure 2.19(*a*)), but if we number the non-terminals in it in the order they were rewritten, we get different numberings, as shown in (*b*) and (*c*).

The sequence of production rules used in leftmost rewriting is called the *leftmost derivation* of a sentence. We do not have to indicate at what position a rule must be applied, nor do we need to give its rule number. Just the alternative is sufficient; the position and the non-terminal are implicit. A *rightmost derivation* is defined in a similar way.

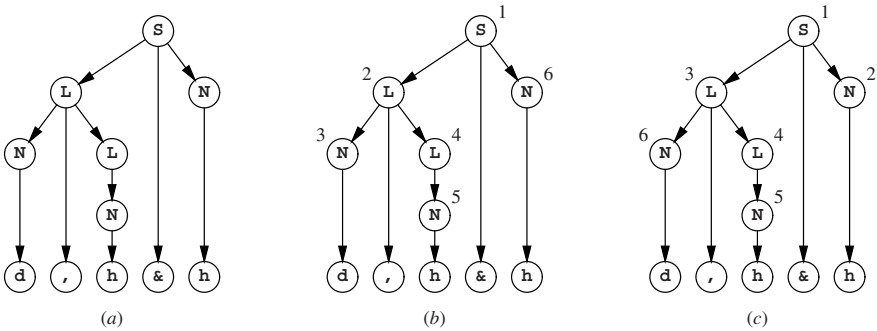


Fig. 2.19. Production tree (*a*) with leftmost (*b*) and rightmost (*c*) derivation order

A leftmost production step can be indicated by using an arrow marked with a small *l*: `N,L&N` $\xrightarrow{l}$ `d,L&N`, and the leftmost production sequence

$$\texttt{S} \xrightarrow{l} \texttt{L\&N} \xrightarrow{l} \texttt{N,L\&N} \xrightarrow{l} \texttt{d,L\&N} \xrightarrow{l} \texttt{d,N\&N} \xrightarrow{l} \texttt{d,h\&N} \xrightarrow{l} \texttt{d,h\&h}$$

can be abbreviated to $\texttt{S} \xrightarrow[l]{*} \texttt{d,h\&h}$. Likewise, the rightmost production sequence

$$\texttt{S} \xrightarrow{r} \texttt{L\&N} \xrightarrow{r} \texttt{L\&h} \xrightarrow{r} \texttt{N,L\&h} \xrightarrow{r} \texttt{N,N\&h} \xrightarrow{r} \texttt{N,h\&h} \xrightarrow{r} \texttt{d,h\&h}$$

can be abbreviated to $\texttt{S} \xrightarrow[r]{*} \texttt{d,h\&h}$. The fact that `S` produces `d,h&h` in any way is written as $\texttt{S} \xrightarrow{*} \texttt{d,h\&h}$.

The task of parsing is to reconstruct the derivation tree (or graph) for a given input string. Some of the most efficient parsing techniques can be understood more easily if viewed as attempts to reconstruct a left- or rightmost derivation process of the input string; the derivation tree then follows automatically. This is why the notion "[left|right]-most derivation" occurs frequently in this book (note the FC grammar used here).

2.5 To Shrink or Not To Shrink

In the previous paragraphs, we have sometimes been explicit as to the question if a right-hand side of a rule may be shorter than its left-hand side and sometimes we have been vague. Type 0 rules may definitely be of the shrinking variety, monotonic rules definitely may not, and Type 2 and 3 rules can shrink only by producing empty (ε); that much is sure.

The original Chomsky hierarchy (Chomsky [385]) was very firm on the subject: only Type 0 rules are allowed to make a sentential form shrink. Type 1, 2 and 3 rules are all monotonic. Moreover, Type 1 rules have to be of the context-sensitive variety, which means that only one of the non-terminals in the left-hand side is actually allowed to be replaced (and then not by ε). This makes for a proper hierarchy in which each next class is a proper subset of its parent and in which all derivation graphs except for those of Type 0 grammars are actually derivation trees.

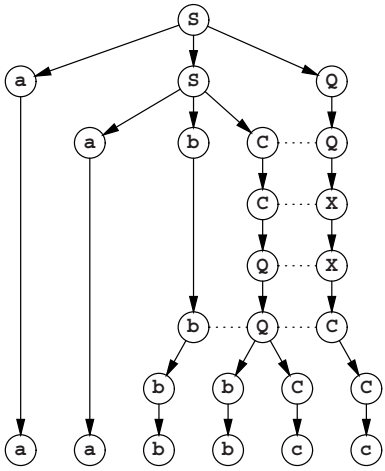
As an example consider the grammar for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ given in Figure 2.7:

```
1.   S_S  →  abc | aSQ
2.  bQc  →  bbcc
3.   cQ  →  Qc
```

which is monotonic but not context-sensitive in the strict sense. It can be made CS by expanding the offending rule 3 and introducing a non-terminal for **c**:

```
1.    S_S  →  abC | aSQ
2.   bQC  →  bbCC
3a.   CQ  →  CX
3b.   CX  →  QX
3c.   QX  →  QC
4.    C  →  c
```

Now the production graph of Figure 2.8 turns into a production tree:



There is an additional reason for shunning ε-rules: they make both proofs and parsers more complicated, sometimes much more complicated; see, for example, Section 9.5.4. So the question arises why we should bother with ε-rules at all; the answer is that they are very convenient for the grammar writer and user.

If we have a language that is described by a CF grammar with ε-rules and we want to describe it by a grammar without ε-rules, then that grammar will almost always be more complicated. Suppose we have a system that can be fed bits of information, like: "Amsterdam is the capital of the Netherlands", "Truffles are expensive", and can then be asked a question. On a very superficial level we can define its input as:

```
input_S:  zero-or-more-bits-of-info question
```

or, in an extended notation

```
input_S:  bit-of-info*  question
```

Since **zero-or-more-bits-of-info** will, among other strings, produce the empty string, at least one of the rules used in its grammar will be an ε-rule; the * in the extended notation already implies an ε-rule somewhere. Still, from the user's point of view, the above definition of input neatly fits the problem and is exactly what we want.

Any attempt to write an ε-free grammar for this input will end up defining a notion that comprises some of the later **bits-of-info** together with the **question** (since the **question** is the only non-empty part, it must occur in all rules involved!) But such a notion does not fit our problem at all and is an artifact:

```
                    input_s:  question-preceded-by-info
question-preceded-by-info:  question
                          | bit-of-info
                                question-preceded-by-info
```

As a grammar becomes more and more complicated, the requirement that it be ε-free becomes more and more of a nuisance: the grammar is working against us, not for us.

This presents no problem from a theoretical point of view: any CF language can be described by an ε-free CF grammar and ε-rules are never needed. Better still, any grammar with ε-rules can be mechanically transformed into an ε-free grammar for the same language. We saw an example of such a transformation above and details of the algorithm are given in Section 4.2.3.1. But the price we pay is that of any grammar transformation: it is no longer our grammar and it reflects the original structure less well.

The bottom line is that the practitioner finds the ε-rule to be a useful tool, and it would be interesting to see if there exists a hierarchy of non-monotonic grammars alongside the usual Chomsky hierarchy. To a large extend there is: Type 2 and Type 3 grammars need not be monotonic (since they can always be made so if the need arises); it turns out that context-sensitive grammars with shrinking rules are equivalent to unrestricted Type 0 grammars; and monotonic grammars with ε-rules are also equivalent to Type 0 grammars. We can now draw the two hierarchies in one picture; see Figure 2.20. Drawn lines separate grammar types with different power. Conceptually different grammar types with the same power are separated by blank space. We see that if we insist on non-monotonicity, the distinction between Type 0 and Type 1 disappears.

A special case arises if the language of a Type 1 to Type 3 grammar itself contains the empty string. This cannot be incorporated into the grammar in the monotonic hierarchy since the start symbol already has length 1 and no monotonic rule can make it shrink. So the empty string has to be attached as a special property to the grammar. No such problem occurs in the non-monotonic hierarchy.

Many parsing methods will in principle work for ε-free grammars only: if something does not produce anything, you can't very well see if it's there. Often the parsing method can be doctored to handle ε-rules, but that invariably increases the complexity of the method. It is probably fair to say that this book would be at least 30%

		Chomsky (monotonic) hierarchy	non-monotonic hierarchy
global production	Type 0	unrestricted phrase structure grammars / monotonic grammars with ε-rules	unrestricted phrase structure grammars
	Type 1	context-sensitive grammars / monotonic grammars, no ε-rules	context-sensitive grammars with non-monotonic rules
local production	Type 2	context-free ε-free grammars	context-free grammars
	Type 3	regular (ε-free) grammars	regular grammars, regular expressions
no production	Type 4	finite-choice	

Fig. 2.20. Summary of grammar hierarchies

thinner if ε-rules did not exist — but then grammars would lose much more than 30% of their usefulness!

2.6 Grammars that Produce the Empty Language

Roughly 1500 years after the introduction of zero as a number by mathematicians in India, the concept is still not well accepted in computer science. Many programming languages do not support records with zero fields, arrays with zero elements, or variable definitions with zero variables; in some programming languages the syntax for calling a routine with zero parameters differs from that for a routine with one or more parameters; many compilers refuse to compile a module that defines zero names; and this list could easily be extended. More in particular, we do not know of any parser generator that can produce a parser for the empty language, the language with zero strings.

All of which brings us to the question of what the grammar for the empty language would look like. First note that the empty language differs from the language that consists of only the empty string, a string with zero characters. This language is easily generated by the grammar $\mathtt{S}_S \rightarrow \varepsilon$, and is handled correctly by the usual *lex-yacc* pipeline. Note that this grammar has no terminal symbols, which means that V_T in Section 2.2 is the empty set.

For a grammar to produce nothing, the production process cannot be allowed to terminate. This suggests one way to obtain such a grammar: $\mathtt{S}_S \rightarrow \mathtt{S}$. This is ugly, however, for two reasons. From an algorithmic point of view the generation process now just loops and no information about the emptiness of the language is obtained; and the use of the symbol S is arbitrary.

Another way is to force the production process to get stuck by not having any production rules in the grammar. Then R in Section 2.2 is empty too, and the form of the grammar is ({S}, {}, S, {}). This is not very satisfactory either, since now we have a non-terminal without a defining rule; and the symbol S is still arbitrary.

A better way is never to allow the production process to get started: have no start symbol. This can be accommodated by allowing a *set* of start symbols in the definition of a grammar rather than a single start symbol. There are other good reasons for doing so. An example is the grammar for a large programming language which has multiple "roots" for module specifications, module definitions, etc. Although these differ at the top level, they have large segments of the grammar in common. If we extend the definition of a CF grammar to use a set of start symbols, the grammar for the empty language obtains the elegant and satisfactory form ({}, {}, {}, {}).

Also on the subject of zero and empty: it might be useful to consider grammar rules in which the *left*-hand side is empty. Terminal productions of the right-hand sides of such rules may appear anywhere in the input, thus modeling noise and other every-day but extraneous events.

Our preoccupation with empty strings, sets, languages, etc. is not frivolous, since it is well known that the ease with which a system handles empty cases is a measure of its cleanliness and robustness.

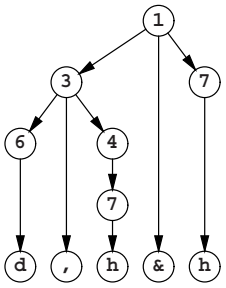
2.7 The Limitations of CF and FS Grammars

When one has been working for a while with CF grammars, one gradually gets the feeling that almost anything could be expressed in a CF grammar. That there are, however, serious limitations to what can be said by a CF grammar is shown by the famous *uvwxy* theorem, which is explained below.

2.7.1 The *uvwxy* Theorem

When we have obtained a sentence from a CF grammar, we may look at each (terminal) symbol in it, and ask: How did it get here? Then, looking at the production tree, we see that it was produced as, say, the n-th member of the right-hand side of rule number m. The left-hand side of this rule, the parent of our symbol, was again produced as the p-th member of rule q, and so on, until we reach the start symbol. We can, in a sense, trace the *lineage* of the symbol in this way. If all rule/member pairs in the lineage of a symbol are different, we call the symbol *original*, and if all the symbols in a sentence are original, we call the sentence "original".

For example, the lineage of the first `h` in the production tree



produced by the grammar

1. S_S → L & N
2. S_S → N
3. L → N , L
4. L → N
5. N → t
6. N → d
7. N → h

is **h** of **7,1** of **4,1** of **3,3** of **1,1**. Here the first number indicates the rule and the second number is the member number in that rule. Since all the rule/member pairs are different the **h** is original.

Now there is only a finite number of ways for a given symbol to be original. This is easy to see as follows. All rule/member pairs in the lineage of an original symbol must be different, so the length of its lineage can never be more than the total number of different rule/member pairs in the grammar. There are only so many of these, which yields only a finite number of combinations of rule/member pairs of this length or shorter. In theory the number of original lineages of a symbol can be very large, but in practice it is very small: if there are more than, say, ten ways to produce a given symbol from a grammar by original lineage, your grammar will be very convoluted indeed!

This puts severe restrictions on original sentences. If a symbol occurs twice in an original sentence, both its lineages must be different: if they were the same, they would describe the same symbol in the same place. This means that there is a maximum length to original sentences: the sum of the numbers of original lineages of all symbols. For the average grammar of a programming language this length is in the order of some thousands of symbols, i.e., roughly the size of the grammar. So, since there is a longest original sentence, there can only be a finite number of original sentences, and we arrive at the surprising conclusion that any CF grammar produces a finite-size kernel of original sentences and (probably) an infinite number of unoriginal sentences!

What do "unoriginal" sentences look like? This is where we come to the *uvwxy* theorem. An unoriginal sentence has the property that it contains at least one symbol in the lineage of which a repetition occurs. Suppose that symbol is a q and the repeated rule is A. We can then draw a picture similar to Figure 2.21, where w is the part produced by the most recent application of A, vwx the part produced by the other application of A and $uvwxy$ is the entire unoriginal sentence. Now we can immediately find another unoriginal sentence, by removing the smaller triangle headed by A and replacing it by a copy of the larger triangle headed by A; see Figure 2.22.

This new tree produces the sentence $uvvwxxy$ and it is easy to see that we can, in this way, construct a complete family of sentences $uv^n wx^n y$ for all $n \geq 0$. This form shows the w nested in a number of v and x brackets, in an indifferent context of u and y.

The bottom line is that when we examine longer and longer sentences in a context-free language, the original sentences become exhausted and we meet only families of closely related sentences telescoping off into infinity. This is summarized

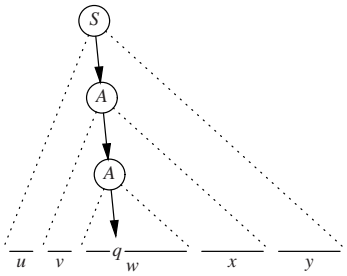


Fig. 2.21. An unoriginal sentence: *uvwxy*

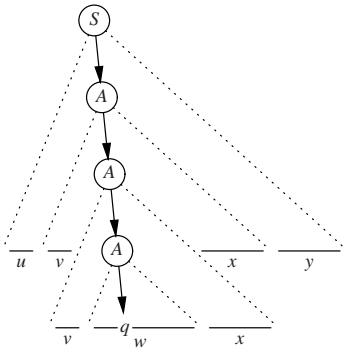


Fig. 2.22. Another unoriginal sentence, uv^2wx^2y

in the *uvwxy* theorem: any sentence generated by a CF grammar that is longer than the longest original sentence from that grammar can be cut into five pieces u, v, w, x and y, in such a way that uv^nwx^ny is a sentence from that grammar for all $n \geq 0$. The *uvwxy* theorem is also called the *pumping lemma for context-free languages* and has several variants.

Two remarks must be made here. The first is that if a language keeps on providing longer and longer sentences without reducing to families of nested sentences, there cannot be a CF grammar for it. We have already encountered the context-sensitive language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ and it is easy to see (but not quite so easy to prove!) that it does not decay into such nested sentences, as sentences get longer and longer. Consequently, there is no CF grammar for it. See Billington [396] for a general technique for such proofs.

The second is that the longest original sentence is a property of the grammar, not of the language. By making a more complicated grammar for a language we can increase the set of original sentences and push away the border beyond which we are forced to resort to nesting. If we make the grammar infinitely complicated, we can push the border to infinity and obtain a phrase structure language from it. How we can make a CF grammar infinitely complicated is described in the section on two-level grammars, 15.2.1.

2.7.2 The *uvw* Theorem

A simpler form of the *uvwxy* theorem applies to regular (Type 3) languages. We have seen that the sentential forms occurring in the production process for a FS grammar all contain only one non-terminal, which occurs at the end. During the production of a very long sentence, one or more non-terminals must occur two or more times, since there are only a finite number of non-terminals. Figure 2.23 shows what we see when we list the sentential forms one by one. The substring *v* has been produced

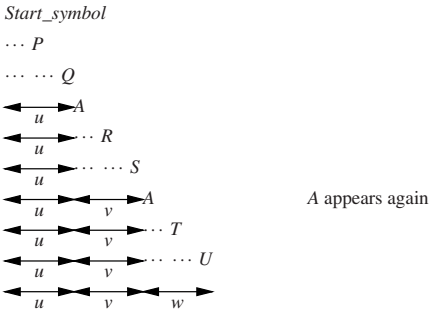


Fig. 2.23. Repeated occurrence of *A* may result in repeated occurrence of *v*

from one occurrence of *A* to the next, *u* is a sequence that allows us to reach *A*, and *w* is a sequence that allows us to terminate the production process. It will be clear that, starting from the second *A*, we could have followed the same path as from the first *A*, and thus have produced *uvvw*. This leads us to the *uvw* theorem, or the *pumping lemma for regular languages*: any sufficiently long string from a regular language can be cut into three pieces *u*, *v* and *w*, so that $uv^n w$ is a string in the language for all $n \geq 0$.

2.8 CF and FS Grammars as Transition Graphs

A *transition graph* is a directed graph in which the arrows are labeled with zero or more symbols from the grammar. The idea is that as you follow the arrows in the graph you produce one of the associated symbols, if there is one, and nothing otherwise. The nodes, often unlabeled, are resting points between producing the symbols. If there is more than one outgoing arrow from a node you can choose any to follow. So the transition graph in Figure 2.24 produces the same strings as the sample grammar on page 23.

It is fairly straightforward to turn a grammar into a set of transition graphs, one for each non-terminal, as Figure 2.25 shows. But it contains arrows marked with non-terminals, and the meaning of "producing" a non-terminal associated with an arrow is not directly clear. Suppose we are at node n_1, from which a transition (arrow) labeled with non-terminal *N* leads to a node n_2, and we want to take that transition.

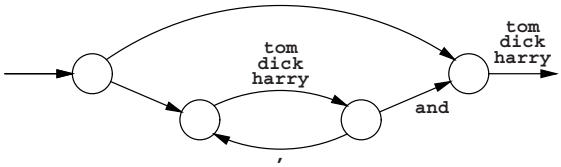


Fig. 2.24. A transition graph for the `[tdh]` language

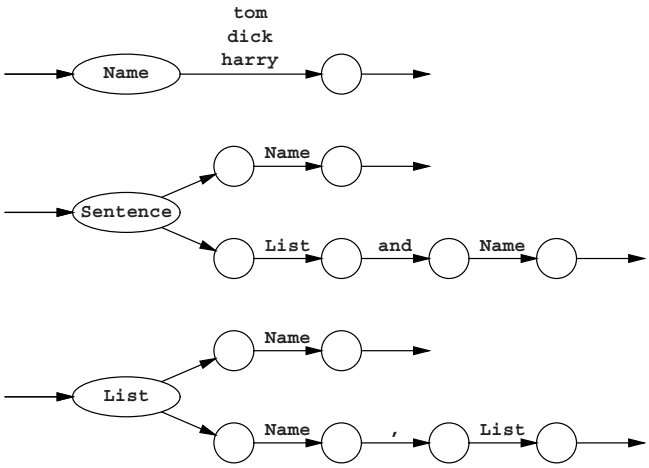


Fig. 2.25. A recursive transition network for the sample grammar on page 23

Rather than producing N by appending it to the output, we push node n_2 on a stack, and continue our walk at the entrance to the transition graph for N. And when we are leaving the transition graph for N, we pop n_2 from the stack and continue at node n_2. This is the *recursive transition network* interpretation of context-free grammars: the set of graphs is the transition network, and the stacking mechanism provides the recursion.

Figure 2.26 shows the right-regular rules of the FS grammar Figure 2.14(a) as transition graphs. Here we have left out the unmarked arrows at the exits of the graphs and the corresponding nodes; we could have done the same in Figure 2.25, but doing so would have complicated the stacking mechanism.

We see that we have to produce a non-terminal only when we are just leaving another, so we do not need to stack anything, and can interpret an arrow marked with a non-terminal N as a jump to the transition graph for N. So a regular grammar corresponds to a (non-recursive) transition network.

If we connect each exit marked N in such a network to the entrance of the graph for N we can ignore the non-terminals, and obtain a transition graph for the corresponding language. When we apply this short-circuiting to the transition network of Figure 2.26 and rearrange the nodes a bit, we get the transition graph of Figure 2.24.

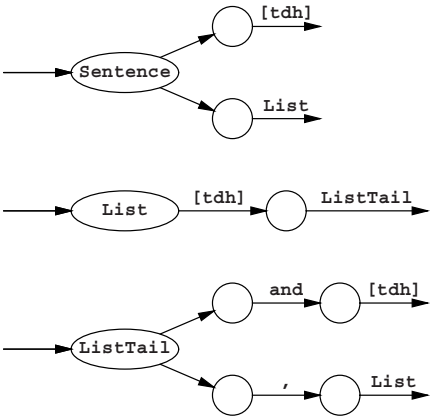


Fig. 2.26. The FS grammar of Figure 2.14(*a*) as transition graphs

2.9 Hygiene in Context-Free Grammars

All types of grammars can contain *useless rules*, rules that cannot play a role in any successful production process. A production process is *successful* when it results in a terminal string. Production attempts can be unsuccessful by getting stuck (no further substitution possible) or by entering a situation in which no substitution sequence will ever remove all non-terminals. An example of a Type 0 grammar that can get stuck is

```
1.    S_S  →  A B
2.     S   →  B A
3.     S   →  C
4.   A B   →  x
5.     C   →  C C
```

When we start with the first rule for `S`, all goes well and we produce the terminal string `x`. But when we start with rule 2 for `S` we get stuck, and when we start with rule 3, we get ourselves in an infinite loop, producing more and more `C`s. Rules 2, 3 and 5 can never occur in a successful production process: they are useless rules, and can be removed from the grammar without affecting the language produced.

Useless rules are not a fundamental problem: they do not obstruct the normal production process. Still, they are dead wood in the grammar, and one would like to remove them. Also, when they occur in a grammar specified by a programmer, they probably point at some error, and one would like to detect them and give warning or error messages.

The problems with the above grammar were easy to understand, but it can be shown that in general it is undecidable whether a rule in a Type 0 or 1 grammar is useless: there cannot be an algorithm that does it correctly in all cases. For context-free grammars the situation is different, however, and the problem is rather easily solved.

Rules in a context-free grammar can be useless through three causes: they may contain undefined non-terminals, they may not be reachable from the start symbol, and they may fail to produce anything. We will now discuss each of these ailments in more detail; an algorithm to rid a grammar of them is given in Section 2.9.5.

2.9.1 Undefined Non-Terminals

The right-hand side of some rule may contain a non-terminal for which no production rule is given. Such a rule will never have issue and can be removed from the grammar. If we do this, we may of course remove the last definition of another non-terminal, which will then in turn become undefined, etc.

We will see further on (for example in Section 4.1.3) that it is occasionally useful to also recognize undefined terminals. Rules featuring them in their right-hand sides can again be removed.

2.9.2 Unreachable Non-Terminals

If a non-terminal cannot be reached from the start symbol, its defining rules will never be used, and it cannot contribute to the production of any sentence. Unreachable non-terminals are sometimes called "unused non-terminals". But this term is a bit misleading, because an unreachable non-terminal A may still occur in some right-hand side $B \rightarrow \cdots A \cdots$, making it look useful, provided B is unreachable; the same applies of course to B, etc.

2.9.3 Non-Productive Rules and Non-Terminals

Suppose X has as its only rule $X \rightarrow aX$ and suppose X can be reached from the start symbol. Now X will still not contribute anything to the sentences of the language of the grammar, since once X is introduced, there is no way to get rid of it: X is a non-productive non-terminal. In addition, any rule which has X in its right-hand side is non-productive. In short, any rule that does not in itself produce a non-empty sublanguage is non-productive. If all rules for a non-terminal are non-productive, the non-terminal is non-productive.

In an extreme case *all* non-terminals in a grammar are non-productive. This happens when all right-hand sides in the grammar contain at least one non-terminal. Then there is just no way to get rid of the non-terminals, and the grammar itself is non-productive.

These three groups together are called *useless non-terminals*.

2.9.4 Loops

The above definition makes "non-useless" all rules that can be involved in the production of a sentence, but there still is a class of rules that are not really useful: rules of the form $A \rightarrow A$. Such rules are called *loops*. Loops can also be indirect: $A \rightarrow B$,

$B \rightarrow C$, $C \rightarrow A$; and they can be hidden: $A \rightarrow PAQ$, $P \xrightarrow{*} \varepsilon$, $Q \xrightarrow{*} \varepsilon$, so a production sequence $A \rightarrow PAQ \rightarrow \ldots A \ldots \rightarrow A$ is possible.

A loop can legitimately occur in the production of a sentence, and if it does, there is also a production of that sentence without the loop. Loops do not contribute to the language and any sentence the production of which involves a loop is *infinitely ambiguous*, meaning that there are infinitely many production trees for it. Algorithms for loop detection are given in Section 4.1.2.

Different parsers react differently to grammars with loops. Some (most of the general parsers) faithfully attempt to construct an infinite number of derivation trees, some (for example, the CYK parser) collapse the loop as described above and some (most deterministic parsers) reject the grammar. The problem is aggravated by the fact that loops can be concealed by ε-rules: a loop may only become visible when certain non-terminals produce ε.

A grammar without useless non-terminals and loops is called a *proper grammar*.

2.9.5 Cleaning up a Context-Free Grammar

Normally, grammars supplied by people do not contain undefined, unreachable or non-productive non-terminals. If they do, it is almost certainly a mistake (or a test!), and we would like to detect and report them. Such anomalies can, however, occur normally in generated grammars or be introduced by some grammar transformations, in which case we wish to detect them to "clean up" the grammar. Cleaning the grammar is also very important when we obtain the result of parsing as a parse-forest grammar (Section 3.7.4, Chapter 13, and many other places).

The algorithm to detect and remove useless non-terminals and rules from a context-free grammar consists of two steps: remove the non-productive rules and remove the unreachable non-terminals. Surprisingly it is not necessary to remove the useless rules due to undefined non-terminals: the first step does this for us automatically.

```
Ss  ->  A B | D E
A   ->  a
B   ->  b C
C   ->  c
D   ->  d F
E   ->  e
F   ->  f D
```

Fig. 2.27. A demo grammar for grammar cleaning

We will use the grammar of Figure 2.27 for our demonstration. It looks fairly innocent: all its non-terminals are defined and it does not exhibit any suspicious constructions.

2.9.5.1 Removing Non-Productive Rules

We find the non-productive rules by finding the productive ones. Our algorithm hinges on the observation that a rule is productive if its right-hand side consists of symbols all of which are productive. Terminal symbols are productive since they produce terminals and empty is productive since it produces the empty string. A non-terminal is productive if there is a productive rule for it, but the problem is that initially we do not know which rules are productive, since that is exactly the thing we are trying to find out.

We solve this dilemma by first marking all rules and non-terminals as "Don't know". We now go through the grammar of Figure 2.27 and for each rule for which we do know that all its right-hand side members are productive, we mark the rule and the non-terminal it defines as "Productive". This yields markings for the rules **A→a**, **C→c**, and **E→e**, and for the non-terminals **A**, **C** and **E**.

Now we know more and apply this knowledge in a second round through the grammar. This allows us to mark the rule **B→bC** and the non-terminal **B**, since now **C** is known to be productive. A third round gives us **S→AB** and **S**. A fourth round yields nothing new, so there is no point in a fifth round.

We now know that **S**, **A**, **B**, **C**, and **E** are productive, but **D** and **F** and the rule **S→DE** are still marked "Don't know". However, now we know more: we know that we have pursued all possible avenues for productivity, and have not found any possibilities for **D**, **F** and the second rule for **S**. That means that we can now upgrade our knowledge "Don't know" to "Non-productive". The rules for **D**, **F** and the second rule for **S** can be removed from the grammar; the result is shown in Figure 2.28. This makes **D** and **F** undefined, but **S** stays in the grammar since it is productive, in spite of having a non-productive rule.

```
S_S  →  A B
 A   →  a
 B   →  b C
 C   →  c
 E   →  e
```

Fig. 2.28. The demo grammar after removing non-productive rules

It is interesting to see what happens when the grammar contains an undefined non-terminal, say *U*. *U* will first be marked "Don't know", and since there is no rule defining it, it will stay "Don't know". As a result, any rule *R* featuring *U* in its right-hand side will also stay "Don't know". Eventually both will be recognized as "Non-productive", and all rules *R* will be removed. We see that an "undefined non-terminal" is just a special case of a "non-productive" non-terminal: it is non-productive because there is no rule for it.

The above knowledge-improving algorithm is our first example of a *closure algorithm*. Closure algorithms are characterized by two components: an *initialization*, which is an assessment of what we know initially, partly derived from the situation

and partly "Don't know"; and an inference rule, which is a rule telling how knowledge from several places is to be combined. The inference rule for our problem was:

> For each rule for which we do know that all its right-hand side members are productive, mark the rule and the non-terminal it defines as "Productive'.'

It is implicit in a closure algorithm that the inference rule(s) are repeated until nothing changes any more. Then the preliminary "Don't know" can be changed into a more definitive "Not X", where "X" was the property the algorithm was designed to detect.

Since it is known beforehand that in the end all remaining "Don't know" indications are going to be changed into "Not X", many descriptions and implementations of closure algorithms skip the whole "Don't know" stage and initialize everything to "Not X". In an implementation this does not make much difference, since the meaning of the bits in computer memory is not in the computer but in the mind of the programmer, but especially in text-book descriptions this practice is unelegant and can be confusing, since it just is not true that initially all the non-terminals in our grammar are "Non-productive".

We will see many examples of closure algorithms in this book; they are discussed in more detail in Section 3.9.

2.9.5.2 Removing Unreachable Non-Terminals

A non-terminal is called *reachable* or *accessible* if there exists at least one sentential form, derivable from the start symbol, in which it occurs. So a non-terminal A is reachable if $S \xrightarrow{*} \alpha A \beta$ for some α and β.

We found the non-productive rules and non-terminals by finding the "productive" ones. Likewise, we find the unreachable non-terminals by finding the reachable ones. For this, we can use the following closure algorithm. First, the start symbol is marked "reachable"; this is the initialization. Then, for each rule in the grammar of the form $A \rightarrow \alpha$ with A marked, all non-terminals in α are marked; this is the inference rule. We continue applying the inference rule until nothing changes any more. Now the unmarked non-terminals are not reachable and their rules can be removed.

The first round marks **A** and **B**; the second marks **C**, and the third produces no change. The result — a clean grammar — is in Figure 2.29. We see that rule **E→e**, which was reachable and productive in Figure 2.27 became isolated by removing the non-productive rules, and is then removed by the second step of the cleaning algorithm.

```
S_S  →  A B
 A   →  a
 B   →  b C
 C   →  c
```

Fig. 2.29. The demo grammar after removing all useless rules and non-terminals

Removing the unreachable rules cannot cause a non-terminal N used in a reachable rule to become undefined, since N can only become undefined by removing all its defining rules but since N is reachable, the above process will not remove any rule for it. A slight modification of the same argument shows that removing the unreachable rules cannot cause a non-terminal N used in a reachable rule to become non-productive: N, which was productive or it would not have survived the previous clean-up step, can only become non-productive by removing some of its defining rules but since N is reachable, the above process will not remove any rule for it. This shows conclusively that after removing non-productive non-terminals and then removing unreachable non-terminals we do not need to run the step for removing non-productive non-terminals again.

It is interesting to note, however, that first removing unreachable non-terminals and then removing non-productive rules may produce a grammar which again contains unreachable non-terminals. The grammar of Figure 2.27 is an example in point.

Furthermore it should be noted that cleaning a grammar may remove *all* rules, including those for the start symbol, in which case the grammar describes the empty language; see Section 2.6.

Removing the non-productive rules is a bottom-up process: only the bottom level, where the terminal symbols live, can know what is productive. Removing unreachable non-terminals is a top-down process: only the top level, where the start symbol(s) live(s), can know what is reachable.

2.10 Set Properties of Context-Free and Regular Languages

Since languages are sets, it is natural to ask if the standard operations on sets — union, intersection, and negation (complement) — can be performed on them, and if so, how.

The *union* of two sets S_1 and S_2 contains the elements that are in either set; it is written $S_1 \cup S_2$.. The *intersection* contains the elements that are in both sets; it is written $S_1 \cap S_2$. And the *negation* of a set S contains those in Σ^* but not in S; it is written $\neg S$. In the context of formal languages the sets are defined through grammars, so actually we want to do the operations on the grammars rather than on the languages.

Constructing the grammar for the union of two languages is trivial for context-free and regular languages (and in fact for all Chomsky types): just construct a new start symbol $S' \rightarrow S_1|S_2$, where S_1 and S_2 are the start symbols of the two grammars that describe the two languages. (Of course, if we want to combine the two grammars into one we must make sure that the names in them differ, but that is easy to do.)

Intersection is a different matter, though, since the intersection of two context-free languages need not be context-free, as the following example shows. Consider the two CF languages $L_1 = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^m$ and $L_2 = \mathtt{a}^m\mathtt{b}^n\mathtt{c}^n$ described by the CF grammars

```
L1s  ->  A P                      L2s  ->  Q C
A    ->  a A b  |  ε     and      Q    ->  a Q  |  ε
P    ->  c P  |  ε                C    ->  b C c  |  ε
```

When we take a string that occurs in both languages and thus in their intersection, it will have the form $\mathbf{a}^p\mathbf{b}^q\mathbf{c}^r$ where $p = q$ because of L_1 and $q = r$ because of L_2. So the intersection language consists of strings of the form $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ and we know that that language is not context-free (Section 2.7.1).

The intersection of CF languages has weird properties. First, the intersection of two CF languages always has a Type 1 grammar — but this grammar is not easy to construct. More remarkably, the intersection of three CF languages is more powerful than the intersection of two of them: Liu and Weiner [390] show that there are languages that can be obtained as the intersection of k CF languages, but not of $k-1$. In spite of that, *any* Type 1 language, and even any Type 0 language, can be constructed by intersecting just two CF languages, provided we are allowed to erase all symbols in the resulting strings that belong to a set of *erasable symbols*.

The CS language we will use to demonstrate this remarkable phenomenon is the set of all strings that consist of two identical parts: *ww*, where *w* is any string over the given alphabet; examples are **aa** and **abbababbab**. The two languages to be intersected are defined by

```
L3s  ->  A P                          L4s  ->  Q C
  A  ->  a A x | b A y | ε     and      Q  ->  a Q | b Q | ε
  P  ->  a P | b P | ε                  C  ->  x C a | y C b | ε
```

where **x** and **y** are the erasable symbols. The first grammar produces strings consisting of three parts, a sequence A_1 of **a**s and **b**s, followed by its "dark mirror" image M_1, in which **a** corresponds to **x** and **b** to **y**, followed by an arbitrary sequence G_1 of **a**s and **b**s. The second grammar produces strings consisting of an arbitrary sequence G_2 of **a**s and **b**s, a "dark" sequence M_2, and its mirror image A_2, in which again **a** corresponds to **x** and **b** to **y**. The intersection forces $A_1 = G_2$, $M_1 = M_2$, and $G_1 = A_2$. This makes A_2 the mirror image of the mirror image of A_1, in other words equal to A_1. An example of a string in the intersection is **abbabyxyyxabbab**, where we see the mirror images **abbab** and **yxyyx**. We now erase the erasable symbols **x** and **y** and obtain our result **abbababbab**.

Using a massive application of the above mirror-mirror trick, one can relatively easily prove that any Type 0 language can be constructed as the intersection of two CF languages, plus a set of erasable symbols. For details see, for example, Révész [394].

Remarkably the intersection of a context-free and a *regular* language is always a context-free language, and, what's more, there is a relatively simple algorithm to construct a grammar for that intersection language. This gives rise to a set of unusual parsing algorithms, which are discussed in Chapter 13.

If we cannot have intersection of two CF languages and stay inside the CF languages, we certainly cannot have negation of a CF language and stay inside the CF languages. If we could, we could negate two languages, take their union, negate the result, and so obtain their intersection. In a formula: $L_1 \cap L_2 = \neg((\neg L_1) \cup (\neg L_2))$; this formula is known as De Morgan's Law.

In Section 5.4 we shall see that union, intersection and negation of regular (Type 3) languages yield regular languages.

It is interesting to speculate what would have happened if formal languages had been based on set theory with all the set operations right from the start, rather than on the Chomsky hierarchy. Would context-free languages still have been invented?

2.11 The Semantic Connection

Sometimes parsing serves only to check the correctness of a string; that the string conforms to a given grammar may be all we want to know, for example because it confirms our hypothesis that certain observed patterns are indeed correctly described by the grammar we have designed for it. Often, however, we want to go further: we know that the string conveys a meaning, its semantics, and this semantics is directly related to the structure of the production tree of the string. (If it is not, we have the wrong grammar!)

Attaching semantics to a grammar is done in a very simple and effective way: to each rule in the grammar, a *semantic clause* is attached which relates the semantics of the members of the right-hand side of the rule to the semantics of the left-hand side, in which case the semantic information flows from the leaves of the tree upwards to the start symbol; or the other way around, in which case the semantic information flows downwards from the start symbol to the leaves; or both ways, in which case the semantic information may have to flow up and down for a while until a stable situation is reached. Semantic information flowing down is called *inherited*: each rule inherits it from its parent in the tree. Semantic information flowing up is called *derived*: each rule derives it from its children.

There are many ways to express semantic clauses. Since our subject is parsing and syntax rather than semantics, we will briefly describe only two often-used and well-studied techniques: attribute grammars and transduction grammars. We shall explain both using the same simple example, the language of sums of one-digit numbers; the semantics of a sentence in this language is the value of the sum. The language is generated by the grammar of Figure 2.30. One of its sentences is, for

```
1.   Sum_S  -> Digit
2.    Sum   -> Sum + Digit
3.  Digit   -> 0 | 1 | ··· | 9
```

Fig. 2.30. A grammar for sums of one-digit numbers

example, `3+5+1`; its semantics is 9.

2.11.1 Attribute Grammars

The semantic clauses in an attribute grammar assume that each node in the production tree has room for one or more *attributes*, which are just values (numbers, strings or anything else) sitting in nodes in production trees. For simplicity we restrict ourselves to attribute grammars with only one attribute per node. The semantic clause

of a rule in such a grammar contains some formulas which compute the attributes of some of the non-terminals in that rule (represented by nodes in the production tree) from those of other non-terminals in that same rule. These *semantic actions* connect only attributes that are local to the rule: the overall semantics is composed as the result of all the local computations.

If the semantic action of a rule R computes the attribute of the left-hand side of R, that attribute is *derived*. If it computes an attribute of one of the non-terminals in the right-hand side of R, say A, then that attribute is *inherited* by A. Derived attributes are also called "synthesized attributes". The attribute grammar for our example is:

1.	`Sum`$_s$	→	`Digit`	$\{A_0 := A_1\}$
2.	`Sum`	→	`Sum + Digit`	$\{A_0 := A_1 + A_3\}$
3a.	`Digit`	→	`0`	$\{A_0 := 0\}$
	...	...		
3j.	`Digit`	→	`9`	$\{A_0 := 9\}$

The semantic clauses are given between curly brackets. A_0 is the (derived) attribute of the left-hand side; $A_1, \ldots, A_n$ are the attributes of the members of the right-hand side. Traditionally, terminal symbols in a right-hand side are also counted in determining the index of A, although they do not (normally) carry attributes; the `Digit` in rule 2 is in position 3 and its attribute is A_3. Most systems for handling attribute grammars have less repetitive ways to express rule 3a through 3j.

The initial production tree for `3+5+1` is given in Figure 2.31. First only the at-

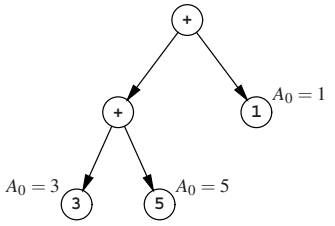


Fig. 2.31. Initial stage of the attributed production tree for `3+5+1`

tributes for the leaves are known, but as soon as all attributes in a right-hand side of a production rule are known, we can use its semantic clause to compute the attribute of its left-hand side. This way the attribute values (semantics) percolate up the tree, finally reach the start symbol and provide us with the semantics of the whole sentence, as shown in Figure 2.32. Attribute grammars are a very powerful method of handling the semantics of a language. They are discussed in more detail in Section 15.3.1.

2.11.2 Transduction Grammars

Transduction grammars define the semantics of a string (the "input string") as another string, the "output string" or "translation", rather than as the final attribute of

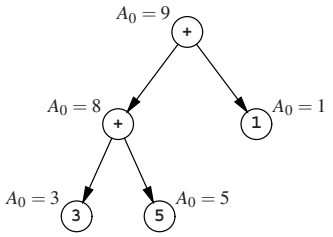


Fig. 2.32. Fully attributed production tree for `3+5+1`

the start symbol. This method is less powerful but much simpler than using attributes and often sufficient. The semantic clause in a production rule is just the string that should be output for the corresponding node. We assume that the string for a node is output just after the strings for all its children. Other variants are possible and in fact usual. We can now write a transduction grammar which translates a sum of digits into instructions to compute the value of the sum.

```
1.    Sum_S  ->  Digit         "make it the result"
2.     Sum   ->  Sum + Digit   "add it to the previous result"
3a.  Digit   ->  0             "take a 0"
       ...   ...
3j.  Digit   ->  9             "take a 9"
```

This transduction grammar translates `3+5+1` into:

```
take a 3
make it the result
take a 5
add it to the previous result
take a 1
add it to the previous result
```

which is indeed what `3+5+1` "means".

2.11.3 Augmented Transition Networks

Semantics can be introduced in a recursive transition network (Section 2.8) by attaching actions to the transitions in the graphs. These actions can set variables, construct data structures, etc. A thus augmented recursive transition network is known as an *Augmented Transition Network* (or *ATN*) (Woods [378]).

2.12 A Metaphorical Comparison of Grammar Types

Text books claim that "Type n grammars are more powerful than Type $n+1$ grammars, for $n = 0, 1, 2$", and one often reads statements like "A regular (Type 3) grammar is not powerful enough to match parentheses". It is interesting to see what kind of power is meant. Naively, one might think that it is the power to generate larger

and larger sets, but this is clearly incorrect: the largest possible set of strings, Σ^*, is easily generated by the Type 3 grammar

$$S_S \rightarrow [\Sigma]\ S \mid \varepsilon$$

where $[\Sigma]$ is an abbreviation for the symbols in the language. It is just when we want to restrict this set, that we need more powerful grammars. More powerful grammars can define more complicated boundaries between correct and incorrect sentences. Some boundaries are so fine that they cannot be described by any grammar (that is, by any generative process).

This idea has been depicted metaphorically in Figure 2.33, in which a rose is approximated by increasingly finer outlines. In this metaphor, the rose corresponds to the language (imagine the sentences of the language as molecules in the rose); the grammar serves to delineate its silhouette. A regular grammar only allows us straight horizontal and vertical line segments to describe the flower; ruler and T-square suffice, but the result is a coarse and mechanical-looking picture. A CF grammar would approximate the outline by straight lines at any angle and by circle segments; the drawing could still be made using the classical tools of compasses and ruler. The result is stilted but recognizable. A CS grammar would present us with a smooth curve tightly enveloping the flower, but the curve is too smooth: it cannot follow all the sharp turns, and it deviates slightly at complicated points; still, a very realistic picture results. An unrestricted phrase structure grammar can represent the outline perfectly. The rose itself cannot be caught in a finite description; its essence remains forever out of our reach.

A more prosaic and practical example can be found in the successive sets of Java[5] programs that can be generated by the various grammar types.

- The set of all lexically correct Java programs can be generated by a regular grammar. A Java program is lexically correct if there are no newlines inside strings, comments are terminated before end-of-file, all numerical constants have the right form, etc.
- The set of all syntactically correct Java programs can be generated by a context-free grammar. These programs conform to the (CF) grammar in the manual.
- The set of all semantically correct Java programs can be generated by a CS grammar. These are the programs that pass through a Java compiler without drawing error messages.
- The set of all Java programs that would terminate in finite time when run with a given input can be generated by an unrestricted phrase structure grammar. Such a grammar would, however, be very complicated, since it would incorporate detailed descriptions of the Java library routines and the Java run-time system.
- The set of all Java programs that solve a given problem (for example, play chess) cannot be generated by a grammar (although the description of the set is finite).

Note that each of the above sets is a subset of the previous set.

[5] We use the programming language Java here because we expect that most of our readers will be more or less familiar with it. Any programming language for which the manual gives a CF grammar will do.

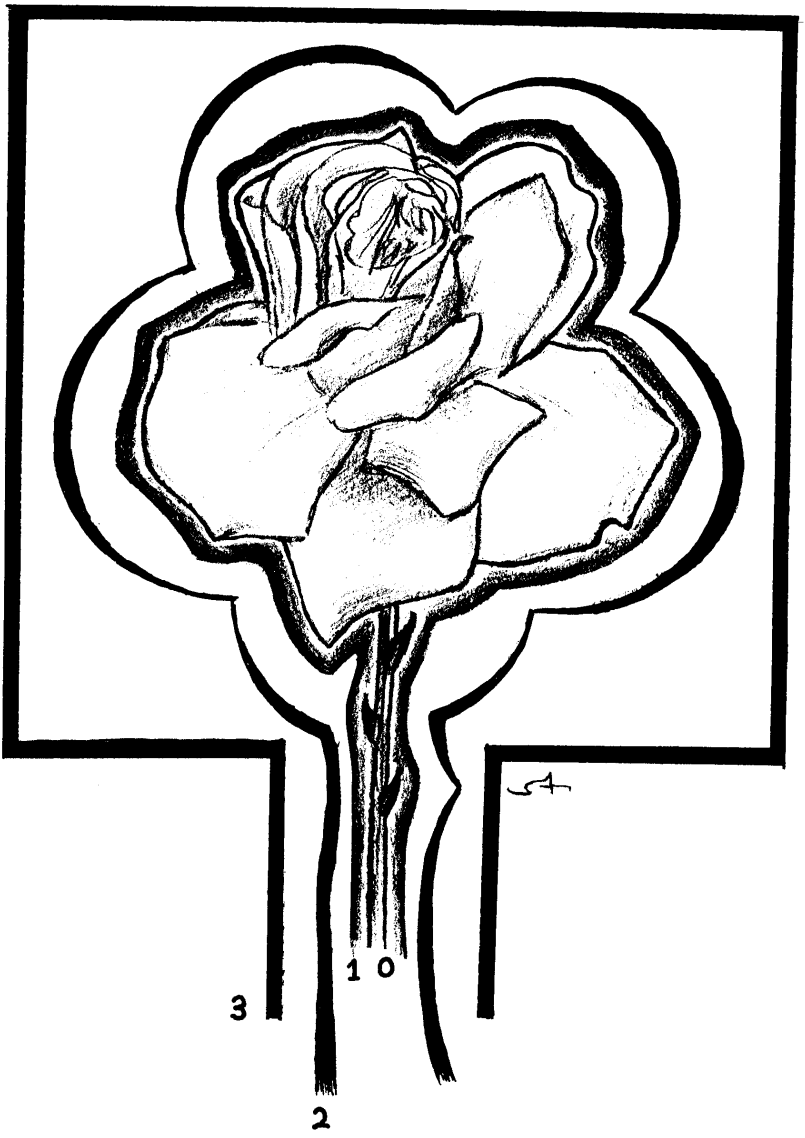


Fig. 2.33. The silhouette of a rose, approximated by Type 3 to Type 0 grammars

2.13 Conclusion

A Chomsky grammar is a finite mechanism that produces a usually infinite set of strings, a "language." Unlike many other set generation mechanisms, this production process assigns a structure to the produced string, which can be utilized to attach semantics to it. For context-free (Type 2) grammars, this structure is a tree, which allows the semantics to be composed from the semantics of the branches. This is the basis of the importance of context-free grammars.

Problems

Problem 2.1: The diagonalization procedure on page 11 seems to be a finite description of a language not on the list. Why is the description not on the list, which contains all finite descriptions after all?

Problem 2.2: In Section 2.1.3.4 we considered the functions n, $n+10$, and $2n$ to find the positions of the bits that should differ from those in line n. What is the general form of these functions, i.e., what set of functions will generate languages that do not have finite descriptions?

Problem 2.3: Write a grammar for Manhattan turtle paths in which the turtle is never allowed to the west of its starting point.

Problem 2.4: Show that the monotonic Type 1 grammar of Figure 2.7 produces all strings of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 1$, and no others. Why is $n = 0$ excluded?

Problem 2.5: Write a Type 1 grammar that produces the language of all strings that consists of two identical parts: ww, where w is any string over the given alphabet (see Section 2.10).

Problem 2.6: On page 34 we have the sentence production mechanism add the newly created sentential forms to the end of the queue, claiming that this realizes breadth-first production. When we put them at the start of the queue, the mechanism uses depth-first production. Show that this does not work.

Problem 2.7: The last paragraph of Section 2.4.1 contains the words "in increasing (actually 'non-decreasing') length". Explain why "non-decreasing" is enough.

Problem 2.8: Relate the number of strings in the finite language produced by a grammar without recursion (page 25) to the structure of that grammar.

Problem 2.9: Refer to Section 2.6. Find more examples in your computing environment where zero as a number gets a second-class treatment.

Problem 2.10: In your favorite parser generator system, write a parser for the language $\{\varepsilon\}$. Same question for the language $\{\}$.

Problem 2.11: Use the uvw theorem (Section 2.7.2) to show that there is no Type 3 grammar for the language $\mathtt{a}^i\mathtt{b}^i$.

Problem 2.12: In Section 2.9 we write that useless rules can be removed from the grammar without affecting the language produced. This seems to suggest that "not affecting the language by its removal" is the actual property we are after, rather than just uselessness. Comment.

Problem 2.13: Write the Chomsky production process of Section 2.2.2 as a closure algorithm.

3

Introduction to Parsing

To parse a string according to a grammar means to reconstruct the production tree (or trees) that indicate how the given string can be produced from the given grammar. It is significant in this respect that one of the first publications on parsing (Greibach's 1963 doctoral thesis [6]), was titled "Inverses of Phrase Structure Generators", where a phrase structure generator is to be understood as a system for producing phrases from a phrase structure (actually context-free) grammar.

Although production of a sentence based on a Type 0 or Type 1 grammar gives rise to a production graph rather than a production tree, and consequently parsing yields a parse graph, we shall concentrate on parsing using a Type 2, context-free grammar, and the resulting parse trees. Occasionally we will touch upon parsing with Type 0 or Type 1 grammars, as for example in Section 3.2, just to show that it *is* a meaningful concept.

3.1 The Parse Tree

There are two important questions on reconstructing the production tree: why do we need it; and how do we do it.

The requirement to recover the production tree is not natural. After all, a grammar is a condensed description of a set of strings, i.e., a language, and our input string either belongs or does not belong to that language; no internal structure or production path is involved. If we adhere to this formal view, the only meaningful question we can ask is if a given string can be recognized according to a grammar; any question as to how would be a sign of senseless, even morbid curiosity. In practice, however, grammars have semantics attached to them: specific semantics is attached to specific rules, and in order to determine the semantics of a string we need to find out which rules were involved in its production and how. In short, recognition is not enough, and we need to recover the production tree to get the full benefit of the syntactic approach.

The recovered production tree is called the *parse tree*. The fact that it is next to impossible to attach semantics to specific rules in Type 0 and Type 1 grammars explains their relative unimportance in parsing, compared to Types 2 and 3.

How we can reconstruct the production tree is the main subject of the rest of this book.

3.1.1 The Size of a Parse Tree

A parse tree for a string of n tokens consists of n nodes belonging to the terminals, plus a number of nodes belonging to the non-terminals. Surprisingly, there cannot be more than $C_G n$ nodes belonging to non-terminals in a parse tree with n token nodes, where C_G is a constant that depends on the grammar, provided the grammar has no loops. This means that the size of any parse tree is linear in the length of the input.

Showing that this is true has to be done in a number of steps. We prove it first for grammars in which all right-hand sides have length 2; these result in *binary trees*, trees in which each node either has two children or is a leaf (a node with no children). Binary trees have the remarkable property that all binary trees with a given number of leaves have the same number of nodes, regardless of their shapes. Next we allow grammars with right-hand sides with lengths > 2, then grammars with unit rules, and finally grammars with nullable rules.

As we said, an input string of length n consists of n token nodes. When the parse tree is not there yet, these nodes are parentless leaves. We are now going to build an arbitrary binary tree to give each of these nodes a parent, labeled with a non-terminal from the grammar. The first parent node P_1 we add lowers the number of parentless nodes by 2, but now P_1 is itself a parentless node; so we now have $n+1$ nodes of which $n-2+1=n-1$ are parentless. The same happens with the second added parent node P_2, regardless of whether one of its children is P_1; so now we have $n+2$ nodes of which $n-2$ are parentless. After j steps we have $n+j$ nodes of which $n-j$ are parentless and after $n-1$ steps we have $2n-1$ nodes of which 1 is parentless. The 1 parentless node is the top node, and the parse tree is complete. So we see that when all right-hand sides have length 2, the parse tree for an input of length n contains $2n-1$ nodes, which is linear in n.

If some of the right-hand sides have length > 2, fewer parent nodes may be required to construct the tree. So the total tree size may be smaller than $2n-1$, which is certainly smaller than $2n$.

If the grammar contains unit rules — rules of the form $A \rightarrow B$ — it is no longer true that adding a parent node reduces the number of parentless nodes: when a parentless node B gets a parent A through the rule $A \rightarrow B$, it is no longer parentless, but the node for A now is, and, what is worse, the number of nodes has gone up by one. And it may be necessary to repeat the process, say with a rule $Z \rightarrow A$, etc. But eventually the chain of unit rules must come to an end, say at P (so we have $P \rightarrow Q \cdots Z \rightarrow A \rightarrow B$), or there would be a loop in the grammar. This means that P gets a parent node with more than one child node and the number of parentless nodes is reduced (or P is the top node). So the worst thing the unit rules can do is

to "lengthen" each node by a constant factor C_u, the maximum length of a unit rule chain, and so the size of the parse tree is smaller than $2C_u n$.

If the grammar contains rules of the form $A \rightarrow \varepsilon$, only a finite number of εs can be recognized between each pair of adjacent tokens in the input, or there would again be a loop in the grammar. So the worst thing nullable rules can do is to "lengthen" the input by a constant factor C_n, the maximum number of εs recognized between two tokens, and the size of the parse tree is smaller than $2C_n C_u n$, which is linear in n.

If, on the other hand, the grammar is allowed to contain loops, both the above processes can introduce unbounded stretches of nodes in the parse tree, which can then reach any size.

3.1.2 Various Kinds of Ambiguity

A sentence from a grammar can easily have more than one production tree, i.e., there can easily be more than one way to produce the sentence. From a formal point of view this is a non-issue (a set does not count how many times it contains an element), but as soon as we are interested in the semantics, the difference becomes significant. Not surprisingly, a sentence with more than one production tree is called *ambiguous*, but we must immediately distinguish between *essential ambiguity* and *spurious ambiguity*. The difference comes from the fact that we are not interested in the production trees per se, but rather in the semantics they describe. An ambiguous sentence is spuriously ambiguous if all its production trees describe the same semantics; if some of them differ in their semantics, the ambiguity is essential. The notion of "ambiguity" can also be defined for grammars: a grammar is essentially ambiguous if it can produce an essentially ambiguous sentence, spuriously ambiguous if it can produce a spuriously ambiguous sentence (but not an essentially ambiguous one) and unambiguous if it cannot do either. For testing the possible ambiguity of a grammar, see Section 9.14.

A simple ambiguous grammar is given in Figure 3.1. Note that rule 2 differs

```
1.    Sum_s  ->  Digit          { A0 := A1 }
2.     Sum   ->  Sum + Sum      { A0 := A1 + A3 }
3a.  Digit   ->  0              { A0 := 0 }
        ...
3j.  Digit   ->  9              { A0 := 9 }
```

Fig. 3.1. A simple ambiguous grammar

from that in Figure 2.30. Now `3+5+1` has two production trees (Figure 3.2) but the semantics is the same in both cases: 9. The ambiguity is spurious. If we change the `+` into a `-`, however, the ambiguity becomes essential, as seen in Figure 3.3. The unambiguous grammar in Figure 2.30 remains unambiguous and retains the correct semantics if `+` is changed into `-`.

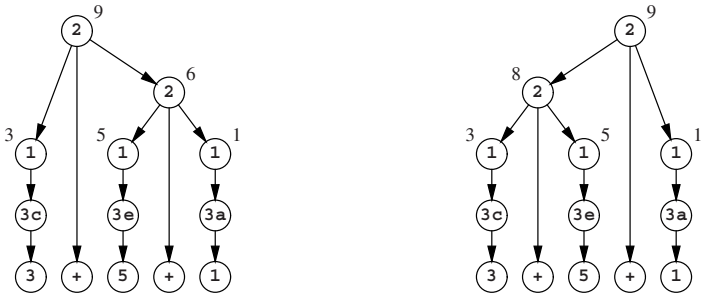


Fig. 3.2. Spurious ambiguity: no change in semantics

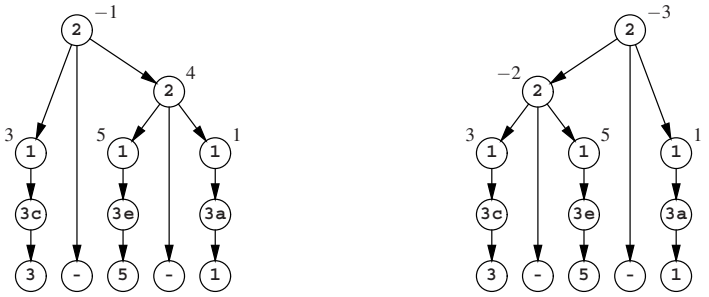


Fig. 3.3. Essential ambiguity: the semantics differ

Strangely enough, languages can also be ambiguous: there are (context-free) languages for which there is no unambiguous grammar. Such languages are *inherently ambiguous*. An example is the language $\mathcal{L} = \mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \cup \mathtt{a}^p\mathtt{b}^p\mathtt{c}^q$. Sentences in $\mathcal{L}$ consist either of a number of **a**s followed by a nested sequence of **b**s and **c**s, or of a nested sequence of **a**s and **b**s followed by a number of **c**s. Example sentences are: **abcc**, **aabbc**, and **aabbcc**; **abbc** is an example of a non-sentence. $\mathcal{L}$ is produced by the grammar of Figure 3.4.

```
S_S  ->  AB | DC
 A   ->  a  | aA
 B   ->  bc | bBc
 D   ->  ab | aDb
 C   ->  c  | cC
```

Fig. 3.4. Grammar for an inherently ambiguous language

Intuitively, it is reasonably clear why $\mathcal{L}$ is inherently ambiguous: any part of the grammar that produces $\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n$ cannot avoid producing $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, and any part of the grammar that produces $\mathtt{a}^p\mathtt{b}^p\mathtt{c}^q$ cannot avoid producing $\mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$. So whatever we do, forms with equal numbers of **a**s, **b**s, and **c**s will always be produced twice. Formally proving that there is really no way to get around this is beyond the scope of this book.

3.1.3 Linearization of the Parse Tree

Often it is inconvenient and unnecessary to construct the actual parse tree: a parser can produce a list of rule numbers instead, which means that it *linearizes* the parse tree. There are three main ways to linearize a tree, prefix, postfix and infix. In *prefix notation*, each node is listed by listing its number followed by prefix listings of the subnodes in left-to-right order; this gives us the leftmost derivation (for the right tree in Figure 3.2):

leftmost: 2 2 1 3c 1 3e 1 3a

If a parse tree is constructed according to this scheme, it is constructed in *pre-order*. In *postfix notation*, each node is listed by listing in postfix notation all the subnodes in left-to-right order, followed by the number of the rule in the node itself; this gives us the rightmost derivation (for the same tree):

rightmost: 3c 1 3e 1 2 3a 1 2

This constructs the parse tree in *post-order*. In *infix notation*, each node is listed by first giving an infix listing between parentheses of the first n subnodes, followed by the rule number in the node, followed by an infix listing between parentheses of the remainder of the subnodes; n can be chosen freely and can even differ from rule to rule, but $n = 1$ is normal. Infix notation is not common for derivations, but is occasionally useful. The case with $n = 1$ is called the *left-corner derivation*; in our example we get:

left-corner: (((3c)1) 2 ((3e)1)) 2 ((3a)1)

The infix notation requires parentheses to enable us to reconstruct the production tree from it. The leftmost and rightmost derivations can do without them, provided we have the grammar ready to find the number of subnodes for each node.

It is easy to tell if a derivation is leftmost or rightmost: a leftmost derivation starts with a rule for the start symbol, while a rightmost derivation starts with a rule that produces terminal symbols only. (If both conditions hold, there is only one rule, which is both a leftmost and a rightmost derivation.)

The existence of several different derivations should not be confused with ambiguity. The different derivations are just notational variants for one and the same production tree. No semantic significance can be attached to their differences.

3.2 Two Ways to Parse a Sentence

The basic connection between a sentence and the grammar it derives from is the parse tree, which describes how the grammar was used to produce the sentence. For the reconstruction of this connection we need a parsing technique. When we consult the extensive literature on parsing techniques, we seem to find dozens of them, yet there are only two techniques to do parsing; all the rest is technical detail and embellishment.

The first method tries to imitate the original production process by rederiving the sentence from the start symbol. This method is called *top-down*, because the parse tree is reconstructed from the top downwards.[1]

The second method tries to roll back the production process and to reduce the sentence back to the start symbol. Quite naturally this technique is called *bottom-up*.

3.2.1 Top-Down Parsing

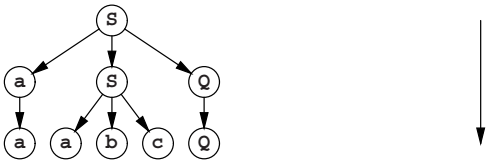
Suppose we have the monotonic grammar for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ from Figure 2.7, which we repeat here:

$$\begin{aligned} \mathtt{S}_S &\rightarrow \mathtt{aSQ} \\ \mathtt{S} &\rightarrow \mathtt{abc} \\ \mathtt{bQc} &\rightarrow \mathtt{bbcc} \\ \mathtt{cQ} &\rightarrow \mathtt{Qc} \end{aligned}$$

and suppose the (input) sentence is **aabbcc**. First we try the top-down parsing method. We know that the production tree must start with the start symbol:

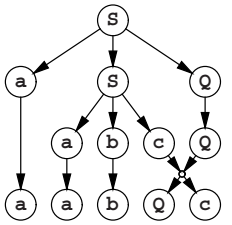
Now what could the second step be? We have two rules for **S**: **S→aSQ** and **S→abc**. The second rule would require the sentence to start with **ab**, which it does not. This leaves us **S→aSQ**:

This gives us a good explanation of the first **a** in our sentence. Again two rules apply: **S→aSQ** and **S→abc**. Some reflection will reveal that the first rule would be a bad choice here: all production rules of **S** start with an **a**, and if we would advance to the stage **aaSQQ**, the next step would inevitably lead to **aaa...**, which contradicts the input string. The second rule, however, is not without problems either:

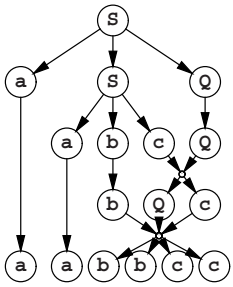


since now the sentence starts with **aabc...**, which also contradicts the input sentence. Here, however, there is a way out: **cQ→Qc**:

[1] Trees grow from their roots downwards in computer science; this is comparable to electrons having a negative charge in physics.



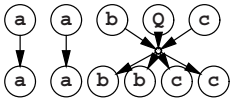
Now only one rule applies: **bQc→bbcc**, and we obtain our input sentence (together with the parse tree):



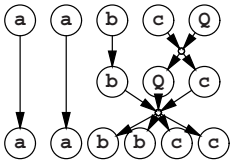
Top-down parsing identifies the production rules (and thus characterizes the parse tree) in prefix order.

3.2.2 Bottom-Up Parsing

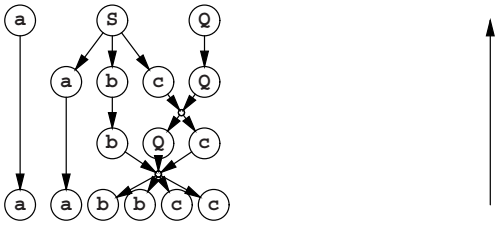
Using the bottom-up technique, we proceed as follows. One production step must have been the last and its result must still be visible in the string. We recognize the right-hand side of **bQc→bbcc** in **aabbcc**. This gives us the final step in the production (and the first in the reduction):



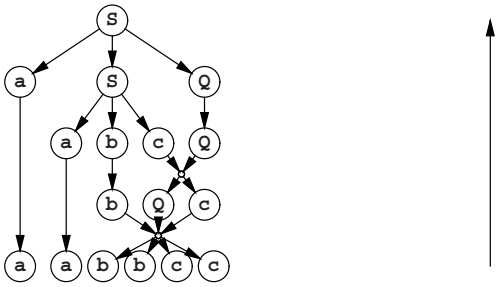
Now we recognize the **Qc** as derived by **cQ→Qc**:



Again we find only one right-hand side: **abc**:



and also our last reduction step leaves us no choice:



Bottom-up parsing tends to identify the production rules in postfix order.

It is interesting to note that bottom-up parsing turns the parsing process into a production process. The above reduction can be viewed as a production with the reversed grammar:

```
 aSQ  →  S
 abc  →  S
bbcc  →  bQc
  Qc  →  cQ
```

augmented with a rule that turns the start symbol into a new terminal symbol:

```
S  →  !
```

and a rule which introduces a new start symbol, the original sentence:

$$\mathtt{I}_S \;\rightarrow\; \mathtt{aabbcc}$$

If, starting from `I`, we can produce `!` we have recognized the input string, and if we have kept records of what we did, we also have obtained the parse tree.

This duality of production and reduction is used by Deussen [21] as a basis for a very fundamental approach to formal languages.

3.2.3 Applicability

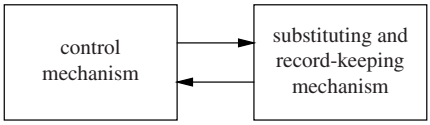
The above examples show that both the top-down and the bottom-up method will work under certain circumstances, but also that sometimes quite subtle considerations are involved, of which it is not at all clear how we can teach them to a computer.

Almost the entire body of parser literature is concerned with formalizing these subtle considerations, and there has been considerable success.

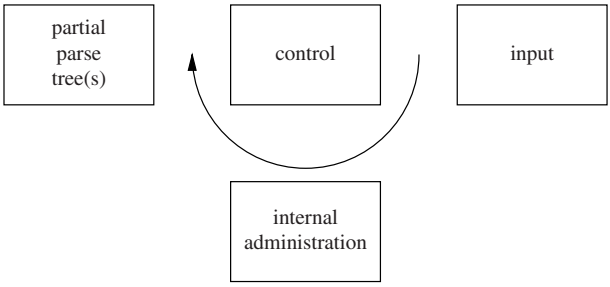
3.3 Non-Deterministic Automata

Both examples above feature two components: a machine that can make substitutions and record a parse tree, and a control mechanism that decides which moves the machine should make. The machine is relatively simple since its substitutions are restricted to those allowed by the grammar, but the control mechanism can be made arbitrarily complex and may incorporate extensive knowledge of the grammar.

This structure can be discerned in all parsing methods: there is always a substituting and record-keeping machine, and a guiding control mechanism:



The substituting machine is called a *non-deterministic automaton* or *NDA*; it is called "non-deterministic" because it often has several possible moves and the particular choice is not predetermined, and an "automaton" because it automatically performs actions in response to stimuli. It manages three components: the input string (actually a copy of it), the partial parse tree and some internal administration. Every move of the NDA transfers some information from the input string through the administration to the partial parse tree. Each of the three components may be modified in the process:



The great strength of an NDA, and the main source of its usefulness, is that it can easily be constructed so that it can only make "correct" moves, that is, moves that keep the system of partially processed input, internal administration and partial parse tree consistent. This has the consequence that we may move the NDA any way we choose: it may move in circles, it may even get stuck, but if it ever gives us an answer, in the form of a finished parse tree, that answer will be correct. It is also essential that the NDA *can* make all correct moves, so that it can produce all parsings if the

control mechanism is clever enough to guide the NDA there. This property of the NDA is also easily arranged.

The inherent correctness of the NDA allows great freedom to the control mechanism, the "control" for short. It may be naive or sophisticated, it may be cumbersome or it may be efficient, it may even be wrong, but it can never cause the NDA to produce an incorrect parsing; and that is a comforting thought. If it is wrong it may, however, cause the NDA to miss a correct parsing, to loop infinitely, or to get stuck in a place where it should not.

3.3.1 Constructing the NDA

The NDA derives directly from the grammar. For a top-down parser its moves consist essentially of the production rules of the grammar and the internal administration is initially the start symbol. The control moves the machine until the internal administration is equal to the input string; then a parsing has been found. For a bottom-up parser the moves consist essentially of the reverse of the production rules of the grammar (see Section 3.2.2) and the internal administration is initially the input string. The control moves the machine until the internal administration is equal to the start symbol; then a parsing has been found. A left-corner parser works like a top-down parser in which a carefully chosen set of production rules has been reversed and which has special moves to undo this reversion when needed.

3.3.2 Constructing the Control Mechanism

Constructing the control of a parser is quite a different affair. Some controls are independent of the grammar, some consult the grammar regularly, some use large tables precomputed from the grammar and some even use tables computed from the input string. We shall see examples of each of these: the "hand control" that was demonstrated at the beginning of this section comes in the category "consults the grammar regularly", backtracking parsers often use a grammar-independent control, LL and LR parsers use precomputed grammar-derived tables, the CYK parser uses a table derived from the input string and Earley's and GLR parsers use several tables derived from the grammar and the input string.

Constructing the control mechanism, including the tables, from the grammar is almost always done by a program. Such a program is called a *parser generator*; it is fed the grammar and perhaps a description of the terminal symbols and produces a program which is a parser. The parser often consists of a driver and one or more tables, in which case it is called *table-driven*. The tables can be of considerable size and of extreme complexity.

The tables that derive from the input string must of course be computed by a routine that is part of the parser. It should be noted that this reflects the traditional setting in which a large number of different input strings is parsed according to a relatively static and unchanging grammar. The inverse situation is not at all unthinkable: many grammars are tried to explain a given input string, for example an observed sequence of events.

3.4 Recognition and Parsing for Type 0 to Type 4 Grammars

Parsing a sentence according to a grammar is in principle always possible provided we know in advance that the string indeed derives from the grammar. If we cannot think of anything better, we can just run the general production process of Section 2.4.1 on the grammar and sit back and wait until the sentence turns up (and we know it will). This by itself is not exactly enough: we must extend the production process a little, so that each sentential form carries its own partial production tree, which must be updated at the appropriate moments, but it is clear that this can be done with some programming effort. We may have to wait a little while (say a couple of million years) for the sentence to show up, but in the end we will surely obtain the parse tree. All this is of course totally impractical, but it still shows us that at least theoretically any string can be parsed *if* we know it is parsable, regardless of the grammar type.

3.4.1 Time Requirements

When parsing strings consisting of more than a few symbols, it is important to have some idea of the *time requirements* of the parser, i.e., the dependency of the time required to finish the parsing on the number of symbols in the input string. Expected lengths of input range from some tens (sentences in natural languages) to some tens of thousands (large computer programs); the length of some input strings may even be virtually infinite (the sequence of buttons pushed on a coffee vending machine over its life-time). The dependency of the time requirements on the input length is also called *time complexity*.

Several characteristic time dependencies can be recognized. A time dependency is *exponential* if each following input symbol multiplies the required time by a constant factor, say 2: each additional input symbol doubles the parsing time. Exponential time dependency is written $O(C^n)$ where C is the constant multiplication factor. Exponential dependency occurs in the number of grains doubled on each field of the famous chess board; this way lies bankruptcy.

A time dependency is *linear* if each following input symbol takes a constant amount of time to process; doubling the input length doubles the processing time. This is the kind of behavior we like to see in a parser; the time needed for parsing is proportional to the time spent on reading the input. So-called *real-time parsers* behave even better: they can produce the parse tree within a constant time after the last input symbol was read; given a fast enough computer they can keep up indefinitely with an input stream of constant speed. Note that this is not necessarily true of linear-time parsers: they can in principle read the entire input of n symbols and then take a time proportional to n to produce the parse tree.

Linear time dependency is written $O(n)$. A time dependency is called *quadratic* if the processing time is proportional to the square of the input length (written $O(n^2)$) and *cubic* if it is proportional to the third power (written $O(n^3)$). In general, a dependency that is proportional to any power of n is called *polynomial* (written $O(n^p)$).

3.4.2 Type 0 and Type 1 Grammars

It is a remarkable result in formal linguistics that the *recognition* problem for a arbitrary Type 0 grammar is unsolvable. This means that there cannot be an algorithm that accepts an arbitrary Type 0 grammar and an arbitrary string and tells us in finite time whether the grammar can produce the string or not. This statement can be proven, but the proof is very intimidating and, what is worse, it does not provide any insight into the cause of the phenomenon. It is a proof by contradiction: we can prove that, if such an algorithm existed, we could construct a second algorithm of which we can prove that it only terminates if it never terminates. Since this is a logical impossibility and since all other premises that went into the intermediate proof are logically sound we are forced to conclude that our initial premise, the existence of a recognizer for Type 0 grammars, is a logical impossibility. Convincing, but not food for the soul. For the full proof see Hopcroft and Ullman [391, pp. 182-183], or Révész [394, p. 98].

It is quite possible to construct a recognizer that works for a certain number of Type 0 grammars, using a certain technique. This technique, however, will not work for all Type 0 grammars. In fact, however many techniques we collect, there will always be grammars for which they do not work. In a sense we just cannot make our recognizer complicated enough.

For Type 1 grammars, the situation is completely different. The seemingly inconsequential property that Type 1 production rules cannot make a sentential form shrink allows us to construct a control mechanism for a bottom-up NDA that will at least work in principle, regardless of the grammar. The internal administration of this control consists of a set of sentential forms that could have played a role in the production of the input sentence; it starts off containing only the input sentence. Each move of the NDA is a reduction according to the grammar. Now the control applies all possible moves of the NDA to all sentential forms in the internal administration in an arbitrary order, and adds each result to the internal administration if it is not already there. It continues doing so until each move on each sentential form results in a sentential form that has already been found. Since no move of the NDA can make a sentential form longer (because all right-hand sides are at least as long as their left-hand sides) and since there are only a finite number of sentential forms as long as or shorter than the input string, this must eventually happen. Now we search the sentential forms in the internal administration for one that consists solely of the start symbol. If it is there, we have recognized the input string; if it is not, the input string does not belong to the language of the grammar. And if we still remember, in some additional administration, how we got this start symbol sentential form, we have obtained the parsing. All this requires a lot of book-keeping, which we are not going to discuss, since nobody does it this way anyway.

To summarize the above, we cannot always construct a parser for a Type 0 grammar, but for a Type 1 grammar we always can. The construction of a practical and reasonably efficient parser for these types of grammars is a very difficult subject on which slow but steady progress has been made during the last 40 years (see (Web)Section 18.1.1). It is not a hot research topic, mainly because Type 0 and Type

1 grammars are well-known to be human-unfriendly and will never see wide application. Yet it is not completely devoid of usefulness, since a good parser for Type 0 grammars would probably make a good starting point for a theorem prover.[2]

The human-unfriendliness consideration does not apply to two-level grammars. Having a practical parser for two-level grammars would be marvelous, since it would allow parsing techniques (with all their built-in automation) to be applied in many more areas than today, especially where context conditions are important. The present possibilities for two-level grammar parsing are discussed in Section 15.2.3.

All known parsing algorithms for Type 0, Type 1 and unrestricted two-level grammars have exponential time dependency.

3.4.3 Type 2 Grammars

Fortunately, much better parsing algorithms are known for CF (Type 2) grammars than for Type 0 and Type 1. Almost all practical parsing is done using CF and FS grammars, and almost all problems in context-free parsing have been solved. The cause of this large difference can be found in the locality of the CF production process: the evolution of one non-terminal in the sentential form is totally independent of the evolution of any other non-terminal, and, conversely, during parsing we can combine partial parse trees regardless of their histories. Neither is true in a context-sensitive grammar.

Both the top-down and the bottom-up parsing processes are readily applicable to CF grammars. In the examples below we shall use the simple grammar

```
Sentence_S  -> Subject Verb Object
Subject     -> the Noun | a Noun | ProperName
Object      -> the Noun | a Noun | ProperName
Verb        -> bit | chased
Noun        -> cat | dog
ProperName  -> ...
```

3.4.3.1 Top-Down CF Parsing

In top-down CF parsing we start with the start symbol and try to produce the input. The keywords here are *predict* and *match*. At any time there is a leftmost non-terminal A in the sentential form and the parser tries systematically to predict a fitting alternative for A, as far as compatible with the symbols found in the input at the position where the result of A should start. This leftmost non-terminal is also called the *goal* of the prediction process.

Consider the example of Figure 3.5, where **`Object`** is the leftmost non-terminal, the "goal". In this situation, the parser will first predict **`the Noun`** for **`Object`**, but will immediately reject this alternative since it requires **`the`** where the input has **`a`**. Next, it will try **`a Noun`**, which is temporarily accepted. The **`a`** is matched and

[2] A theorem prover is a program that, given a set of axioms and a theorem, proves or disproves the theorem without or with minimal human intervention.

Input:	**the**	**cat**	**bit**	**a dog**
Sentential form:	**the**	**cat**	**bit**	**Object**
(the internal administration)				

Fig. 3.5. Top-down parsing as the imitation of the production process

the new leftmost non-terminal is **Noun**. This parse will succeed when **Noun** eventually produces **dog**. The parser will then attempt a third prediction for **Object**, **ProperName**; this alternative is not immediately rejected as the parser cannot see that **ProperName** cannot start with **a**. It will fail at a later stage.

There are two serious problems with this approach. Although it can, in principle, handle arbitrary CF grammars, it will loop on some grammars if implemented naively. This can be avoided by using some special techniques, which result in general top-down parsers; these are treated in detail in Chapter 6. The second problem is that the algorithm requires exponential time since any of the predictions may turn out wrong and may have to be corrected by trial and error. The above example shows that some efficiency can be gained by preprocessing the grammar: it is advantageous to know in advance what tokens can start **ProperName**, to avoid predicting an alternative that is doomed in advance. This is true for most non-terminals in the grammar and this kind of information can be easily computed from the grammar and stored in a table for use during parsing. For a reasonable set of grammars, linear time dependency can be achieved, as explained in Chapter 8.

3.4.3.2 Bottom-Up CF Parsing

In bottom-up CF parsing we start with the input and try to reduce it to the start symbol. Here the keywords are *shift* and *reduce*. When we are in the middle of the process, we have in our hands a sentential form reduced from the input. Somewhere in this sentential form there must be a segment (a substring) that was the result of the last production step that produced this sentential form. This segment corresponds to the right-hand side α of a production rule $A \rightarrow \alpha$ and must now be reduced to A. The segment and the production rule together are called the *handle* of the sentential form, a quite fitting expression; see Figure 3.6. (When the production rule is obvious

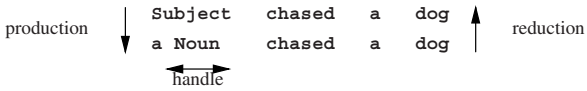


Fig. 3.6. Bottom-up parsing as the inversion of the production process

from the way the segment was found, the matching segment alone is often called the "handle". We will usually follow this custom, but call the matching segment the *handle segment* when we feel that that is clearer.)

The trick is to find the handle. It must be the right-hand side of a rule, so we start looking for such a right-hand side by shifting symbols from the sentential form

into the internal administration. When we find a right-hand side we reduce it to its left-hand side and repeat the process, until only the start symbol is left. We will not always find the correct handle this way; if we err, we will get stuck further on, will have to undo some steps, shift in more symbols and try again. In Figure 3.6 we could have reduced the **`a Noun`** to **`Object`**, thereby boldly heading for a dead end.

There are essentially the same two problems with this approach as with the top-down technique. It may loop, and will do so on grammars with ε-rules: it will continue to find empty productions all over the place. This can be remedied by touching up the grammar. And it can take exponential time, since the correct identification of the handle may have to be done by trial and error. Again, doing preprocessing on the grammar often helps: it is easy to see from the grammar that **`Subject`** can be followed by **`chased`**, but **`Object`** cannot. So it is unprofitable to reduce a handle to **`Object`** if the next symbol is **`chased`**.

3.4.4 Type 3 Grammars

A right-hand side in a regular grammar contains at most one non-terminal, so there is no difference between leftmost and rightmost derivation. Top-down methods are much more efficient for right-regular grammars; bottom-up methods work better for left-regular grammars. When we take the production tree of Figure 2.15 and if we turn it 45° counterclockwise, we get the production chain of Figure 3.7. The se-

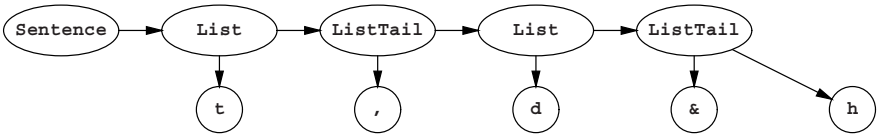


Fig. 3.7. The production tree of Figure 2.15 as a production chain

quence of non-terminals rolls on to the right, producing terminal symbols as they go. In parsing, we are given the terminal symbols and are supposed to construct the sequence of non-terminals. The first one is given, the start symbol (hence the preference for top-down). If only one rule for the start symbol starts with the first symbol of the input we are lucky and know which way to go. Very often, however, there are many rules starting with the same symbol and then we are in need of more wisdom. As with Type 2 grammars, we can of course find the correct continuation by trial and error, but far more efficient methods exist that can handle any regular grammar. Since they form the basis of some advanced parsing techniques, they are treated separately, in Chapter 5.

3.4.5 Type 4 Grammars

Finite-choice (FC) grammars do not involve production trees, and membership of a given input string in the language of the FC grammar can be determined by simple

look-up. This look-up is generally not considered to be "parsing", but is still mentioned here for two reasons. First, it can benefit from parsing techniques, and second, it is often required in a parsing environment. Natural languages have some categories of words that have only a very limited number of members; examples are the pronouns, the prepositions and the conjunctions. It is often important to decide quickly if a given word belongs to one of these finite-choice categories or will have to be analysed further. The same applies to reserved words in a programming language.

One approach is to consider the FC grammar as a regular grammar and apply the techniques of Chapter 5. This is often amazingly efficient.

Another often-used approach is to use a hash table. See any book on algorithms, for example Cormen et al. [415], or Goodrich and Tamassia [416].

3.5 An Overview of Context-Free Parsing Methods

Among the Chomsky grammar types the context-free (Type 2) grammars occupy the most prominent position. This has three reasons: 1. CF parsing results in trees, which allow semantics to be expressed and combined easily; 2. CF languages cover a large part of the languages one would like to process automatically; 3. efficient CF parsing is possible – though sometimes with great difficulty. The context-free grammars are followed immediately by the finite-state grammars in importance. This is because the world and especially equipment is finite; vending machines, remote controls, virus detectors, all exhibit finite-state behavior. The rest of the chapters in this book will therefore be mainly concerned with CF parsing, with the exception of Chapter 5 (finite-state grammars) and Chapter 15 (non-Chomsky systems). We shall now give an overview of the context-free parsing methods.

The reader of literature about parsing is confronted with a large number of techniques with often unclear interrelationships. Yet all techniques can be placed in a single framework, according to some simple criteria; they are summarized in Figure 3.11.

We have already seen that a parsing technique is either top-down, reproducing the input string from the start symbol, or bottom-up, reducing the input to the start symbol. The next division is that between directional and non-directional parsing methods.

3.5.1 Directionality

Non-directional methods construct the parse tree while accessing the input in any order they see fit. This of course requires the entire input to be in memory before parsing can start. There is a top-down and a bottom-up version. *Directional* parsers access the input tokens one by one in order, all the while updating the partial parse tree(s). There is again a top-down and a bottom-up version.

3.5.1.1 Non-Directional Methods

The non-directional top-down method is simple and straightforward and has probably been invented independently by many people. To the best of our knowledge it was first described by Unger [12] in 1968, but in his article he gives the impression that the method already existed. The method has not received much attention in the literature but is more important than one might think, since it is used anonymously in a number of other parsers. We shall call it Unger's method; it is described in Section 4.1.

The non-directional bottom-up method has also been discovered independently by a number of people, among whom Cocke (in Hays [3, Sect. 17.3.1]), Younger [10], and Kasami [13]; an earlier description is by Sakai [5]. It is named CYK (or sometimes CKY) after the three best-known inventors. It has received considerable attention since its naive implementation is much more efficient than that of Unger's method. The efficiency of both methods can be improved, however, arriving at roughly the same performance; see Sheil [20]. The CYK method is described in Section 4.2.

Non-directional methods usually first construct a data structure which summarizes the grammatical structure of the input sentence. Parse trees can then be derived from this data structure in a second stage.

3.5.1.2 Directional Methods

The directional methods process the input symbol by symbol, from left to right. (It is also possible to parse from right to left, using a mirror image of the grammar; this is occasionally useful.) This has the advantage that parsing can start, and indeed progress, considerably before the last symbol of the input is seen. The directional methods are all based explicitly or implicitly on the parsing automaton described in Section 3.4.3, where the top-down method performs predictions and matches and the bottom-up method performs shifts and reduces.

Directional methods can usually construct the (partial) parse tree as they proceed through the input string, unless the grammar is ambiguous and some postprocessing may be required.

3.5.2 Search Techniques

A third way to classify parsing techniques concerns the search technique used to guide the (non-deterministic!) parsing automaton through all its possibilities to find one or all parsings.

There are in general two methods for solving problems in which there are several alternatives in well-determined points: *depth-first search*, and *breadth-first search*.

- In depth-first search we concentrate on one half-solved problem. If the problem bifurcates at a given point P, we store one alternative for later processing and keep concentrating on the other alternative. If this alternative turns out to be a

failure (or even a success, but we want all solutions), we roll back our actions to point *P* and continue with the stored alternative. This is called *backtracking*.

- In breadth-first search we keep a set of half-solved problems. From this set we compute a new set of (better) half-solved problems by examining each old half-solved problem; for each alternative, we create a copy in the new set. Eventually, the set will come to contain all solutions.

Depth-first search has the advantage that it requires an amount of memory that is proportional to the size of the problem, unlike breadth-first search, which may require exponential memory. Breadth-first search has the advantage that it will find the simplest solution first. Both methods require in principle exponential time. If we want more efficiency (and exponential requirements are virtually unacceptable), we need some means to restrict the search. See any book on algorithms, for example Sedgewick [417] or Goodrich and Tamassia [416], for more information on search techniques.

These search techniques are not at all restricted to parsing and can be used in a wide array of contexts. A traditional one is that of finding an exit from a maze. Figure 3.8(*a*) shows a simple maze with one entrance and two exits. Figure 3.8(*b*) depicts the path a depth-first search will take; this is the only option for the human maze-walker: he cannot duplicate himself and the maze. Dead ends make the depth-first search backtrack to the most recent untried alternative. If the searcher will also backtrack at each exit, he will find all exits. Figure 3.8(*c*) shows which rooms are

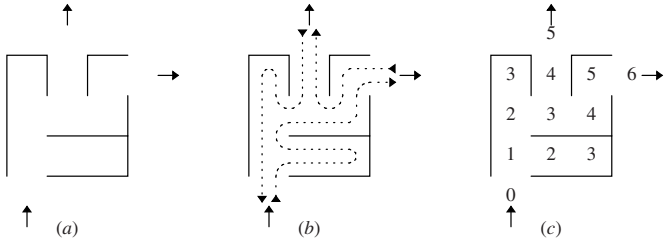


Fig. 3.8. A simple maze with depth-first and breadth-first visits

examined in each stage of the breadth-first search. Dead ends (in stage 3) cause the search branches in question to be discarded. Breadth-first search will find the shortest way to an exit (the shortest solution) first. If it continues until there are no branches left, it will find all exits (all solutions).

3.5.3 General Directional Methods

The idea that parsing is the reconstruction of the production process is especially clear when using a directional method. It is summarized in the following two sound bites.

A directional top-down (left-to-right) CF parser identifies leftmost productions in production order.

and

A directional bottom-up (left-to-right) CF parser identifies rightmost productions in reverse production order.

We will use the very simple grammar

$$\begin{array}{lcl} \mathtt{S_S} & \rightarrow & \mathtt{P\ Q\ R} \\ \mathtt{P} & \rightarrow & \mathtt{p} \\ \mathtt{Q} & \rightarrow & \mathtt{q} \\ \mathtt{R} & \rightarrow & \mathtt{r} \end{array}$$

to demonstrate this. The grammar produces only one string, **pqr**.

The leftmost production process for **pqr** proceeds as follows:

$$\begin{array}{ll} & \mathtt{|S} \\ 1 & \mathtt{|P\ Q\ R} \\ 2 & \mathtt{p\ |Q\ R} \\ 3 & \mathtt{p\ q\ |R} \\ 4 & \mathtt{p\ q\ r\ |} \end{array}$$

where the | indicates how far the production process has proceeded. The top-down analysis mimics this process by first identifying the rule that produced the **p**, **P→p**, then the one for **q**, etc.:

$$\mathtt{|S} \overset{\mathtt{S\rightarrow PQR}}{\Rightarrow} \mathtt{|PQR} \overset{\mathtt{P\rightarrow p}}{\Rightarrow} \mathtt{p|QR} \overset{\mathtt{Q\rightarrow q}}{\Rightarrow} \mathtt{pq|R} \overset{\mathtt{R\rightarrow r}}{\Rightarrow} \mathtt{pqr|}$$

(1) (2) (3) (4)

The rightmost production process for **pqr** proceeds as follows:

$$\begin{array}{ll} & \mathtt{S|} \\ 1 & \mathtt{P\ Q\ R|} \\ 2 & \mathtt{P\ Q|\ r} \\ 3 & \mathtt{P|\ q\ r} \\ 4 & \mathtt{|\ p\ q\ r} \end{array}$$

where the | again indicates how far the production process has proceeded. The bottom-up analysis rolls back this process. To do this, it must first identify the rule in production step 4, **P→p** and use it as a reduction, then step 3, **Q→q**, etc. Fortunately the parser can easily do this, because the rightmost production process makes the boundary between the unprocessed and the processed part of the sentential form creep to the left, so the last production brings it to the left end of the result, as we see above. The parsing process can then start picking it up there:

$$\mathtt{|pqr} \overset{\mathtt{P\rightarrow p}}{\Rightarrow} \mathtt{P|qr} \overset{\mathtt{Q\rightarrow q}}{\Rightarrow} \mathtt{PQ|r} \overset{\mathtt{R\rightarrow r}}{\Rightarrow} \mathtt{pqr|} \overset{\mathtt{S\rightarrow pqr}}{\Rightarrow} \mathtt{S|}$$

(4) (3) (2) (4)

This double reversal is inherent in directional bottom-up parsing.

The connection between parse trees under construction and sentential forms is shown in Figure 3.9, where the dotted lines indicate the sentential forms. On the left

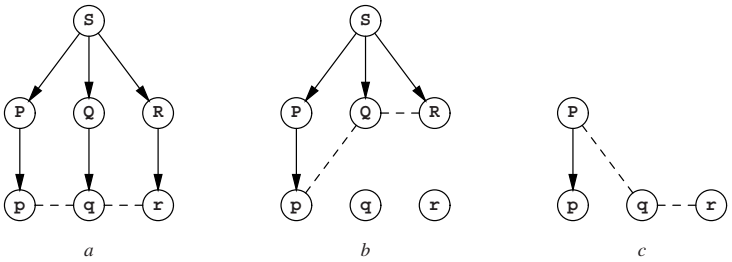


Fig. 3.9. Sentential forms in full parse tree (*a*), during top-down (*b*), and during bottom-up (*c*)

we have the complete parse tree; the corresponding sentential form is the string of terminals. The middle diagram shows the partial parse tree in a top-down parser after the **p** has been processed. The sentential form corresponding to this situation is **pQR**. It resulted from the two productions **S** ⇒ **PQR** ⇒ **pQR**, which gave rise to the partial parse tree. The diagram on the right shows the partial parse tree after the **p** has been processed in a bottom-up parser. The corresponding sentential form is **Pqr**, resulting from **pqr** ⇐ **Pqr**; the single reduction gave rise to a partial parse tree of only one node.

Combining depth-first or breadth-first with top-down or bottom-up gives four classes of parsing techniques. The top-down techniques are treated in Chapter 6. The depth-first top-down technique allows a very simple implementation called recursive descent; this technique, which is explained in Section 6.6, is very suitable for writing parsers by hand. Since depth-first search is built into the Prolog language, recursive descent parsers for a large number of grammars can be formulated very elegantly in that language, using a formalism called "Definite Clause Grammars" (Section 6.7). The applicability of this technique can be extended to cover all grammars by using a device called "cancellation" (Section 6.8).

The bottom-up techniques are treated in Chapter 7. The combination of breadth-first and bottom-up leads to the class of Earley parsers, which have among them some very effective and popular parsers for general CF grammars (Section 7.2). A formally similar but implementationwise quite different approach leads to "chart parsing" (Section 7.3).

Sudkamp [397, Chapter 4] gives a full formal explanation of [breadth-first | depth-first][top-down | bottom-up] context-free parsing.

3.5.4 Linear Methods

Most of the general search methods indicated in the previous section have exponential time dependency in the worst case: each additional symbol in the input multiplies the parsing time by a constant factor. Such methods are unusable except for very small input length, where 20 symbols is about the maximum. Even the best variants of the above methods require cubic time in the worst case: for 10 tokens they perform 1000 actions, for 100 tokens 1 000 000 actions and for 10 000 tokens (a fair-sized

computer program file) 10^{12} actions, which even at 10 nanoseconds per action will already take almost 3 hours. It is clear that for real speed we should like to have a linear-time general parsing method. Unfortunately no such method has been discovered to date, and although there is no proof that such a method could not exist, there are strong indications that that is the case; see Section 3.10 for details. Compare this to the situation around unrestricted phrase structure parsing, where it has been proved that no algorithm for it can exist (see Section 3.4.2).

So, in the meantime, and probably forever, we shall have to drop one of the two adjectives from our goal, a linear-time general parser. We can have a general parser, which will need cubic time at best, or we can have a linear-time parser, which will not be able to handle all CF grammars, but not both. Fortunately there are linear-time parsing methods (in particular LR parsing) that can handle very large classes of grammars but still, a grammar that is designed without regard for a parsing method and just describes the intended language in the most natural way has a small chance of allowing linear parsing automatically. In practice, grammars are often first designed for naturalness and then adjusted by hand to conform to the requirements of an existing parsing method. Such an adjustment is usually relatively simple, depending on the parsing method chosen. In short, making a linear-time parser for an arbitrary given grammar is 10% hard work; the other 90% can be done by computer.

We can achieve linear parsing time by restricting the number of possible moves of our non-deterministic parsing automaton to one in each situation. Since the moves of such an automaton involve no choice, it is called a "deterministic automaton".

The moves of a deterministic automaton are determined unambiguously by the input stream (we can speak of a stream now, since the automaton operates from left to right). A consequence of this is that a deterministic automaton can give only one parsing for a sentence. This is all right if the grammar is unambiguous, but if it is not, the act of making the automaton deterministic has pinned us down to one specific parsing. We shall say more about this in Sections 8.2.5.3 and 9.9.

All that remains is to explain how a deterministic control mechanism for a parsing automaton can be derived from a grammar. Since there is no single good solution to the problem, it is not surprising that quite a number of sub-optimal solutions have been found. From a very global point of view they all use the same technique: they analyse the grammar in depth to bring to the surface information that can be used to identify dead ends. These are then closed. If the method, applied to a grammar, closes enough dead ends so that no choices remain, the method succeeds for that grammar and gives us a linear-time parser. Otherwise it fails and we either have to look for a different method or adapt our grammar to the method.

A (limited) analogy with the maze problem can perhaps make this clearer. If we are allowed to do preprocessing on the maze (unlikely but instructive) the following method will often make our search through it deterministic. We assume that the maze consists of a grid of square rooms, as shown in Figure 3.10(*a*). Depth-first search would find a passage through the maze in 13 moves (Figure 3.10(*b*)). Now we preprocess the maze as follows: if there is a room with three walls, add the fourth wall, and continue with this process until no rooms with three walls are left. If all rooms now have either two or four walls, there are no choices left and our method

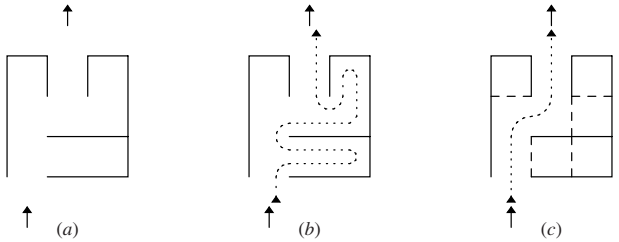


Fig. 3.10. A single-exit maze made deterministic by preprocessing

has succeeded; see Figure 3.10(*c*), where the passage now takes 5 moves, with no searching involved. We see how this method brings information about dead ends to the surface, to help restrict the choice.

It should be pointed out that the above analogy is a limited one. It is concerned with only one object, the maze, which is preprocessed. In parsing we are concerned with two objects, the grammar, which is static and can be preprocessed, and the input, which varies. (But see Problem 3.6 for a way to extend the analogy.)

Returning to the parsing automaton, we can state the fact that it is deterministic more precisely: a parsing automaton is *deterministic* with *look-ahead k* if its control mechanism can, given the internal administration and the next *k* symbols of the input, decide unambiguously what to do next — to either match or predict and what to predict in the top-down case, and to either shift or reduce and how to reduce in the bottom-up case.

It stands to reason that a deterministic automaton creates a linear-time parser, but this is not completely obvious. The parser may know in finite time what to do in each step, but many steps may have to be executed for a given input token. More specifically, some deterministic techniques can require *k* steps for a given position *k*, which suggests that quadratic behavior is possible (see Problem 3.5). But each parsing step either creates a node (predict and reduce) or consumes an input token (match and shift). Both actions can only be performed $O(n)$ times where n is the length of the input: the first because the size of the parse tree is only $O(n)$ and the second because there are only n input tokens. So however the actions of the various tasks are distributed, their total cannot exceed $O(n)$.

Like grammar types, deterministic parsing methods are indicated by initials, like LL, LALR, etc. If a method X uses a look-ahead of k symbols it is called $X(k)$. All deterministic methods require some form of grammar preprocessing to derive the parsing automaton, plus a parsing algorithm or driver to process the input using that automaton.

3.5.5 Deterministic Top-Down and Bottom-Up Methods

There is only one deterministic top-down method; it is called *LL*. The first L stands for Left-to-right, the second for "identifying the Leftmost production", as directional top-down parsers do. LL parsing is treated in Chapter 8. LL parsing, especially LL(1)

is very popular. LL(1) parsers are often generated by a parser generator but a simple variant can, with some effort, be written by hand, using recursive-descent techniques; see Section 8.2.6. Occasionally, the LL(1) method is used starting from the last token of the input backwards; it is then called *RR(1)*.

There are quite a variety of deterministic bottom-up methods, the most powerful being called *LR*, where again the L stands for Left-to-right, and the R stands for "identifying the Rightmost production". Linear bottom-up methods are treated in Chapter 9. Their parsers are invariably generated by a parser generator: the control mechanism of such a parser is so complicated that it is not humanly possible to construct it by hand. Some of the deterministic bottom-up methods are very popular and are perhaps used even more widely than the LL(1) method.

LR(1) parsing is more powerful than LL(1) parsing, but also more difficult to understand and less convenient. The other methods cannot be compared easily to the LL(1) method. See Chapter 17.1 for a comparison of practical parsing methods. The LR(1) method can also be applied backwards and is then called *RL(1)*.

Both methods use look-ahead to determine which actions to take. Usually this look-ahead is restricted to one token (LL(1), LR(1), etc.) or a few tokens at most, but it is occasionally helpful to allow unbounded look-ahead. This requires different parsing techniques, which results in a subdivision of the class of deterministic parsers; see Figure 3.11.

The great difference in variety between top-down and bottom-up methods is easily understood when we look more closely at the choices the corresponding parsers face. A top-down parser has by nature little choice: if a terminal symbol is predicted, it has no choice and can only ascertain that a match is present; only if a non-terminal is predicted does it have a choice in the production of that non-terminal. A bottom-up parser can always shift the next input symbol, even if a reduction is also possible (and it often has to do so). If, in addition, a reduction is possible, it may have a choice between a number of right-hand sides. In general it has more choice than a top-down parser and more powerful methods are needed to make it deterministic.

3.5.6 Non-Canonical Methods

For many practical grammars the above methods still do not yield a linear-time deterministic parser. One course of action that is often taken is to modify the grammar slightly so as to fit it to the chosen method. But this is unfortunate because the resulting parser then yields parse trees that do not correspond to the original grammar, and patching up is needed afterwards. Another alternative is to design a parser so it postpones the decisions it cannot take for lack of information and continue parsing "at half power" until the information becomes available. Such parsers are called *non-canonical* because they identify the nodes in the parse trees in non-standard, "non-canonical" order. Needless to say this requires treading carefully, and some of the most powerful, clever, and complicated deterministic parsing algorithms come in this category. They are treated in Chapter 10.

3.5.7 Generalized Linear Methods

When our attempt to construct a deterministic control mechanism fails and leaves us with a non-deterministic but almost deterministic one, we need not despair yet: we can fall back on breadth-first search to solve the remnants of non-determinism at run-time. The better our original method was, the less non-determinism will be left, the less often breadth-first search will be needed, and the more efficient our parser will be. Such parsers are called "generalized parsers"; generalized parsers have been designed for most of the deterministic methods, both top-down and bottom-up. They are described in Chapter 11. Generalized LR (or GLR) (Tomita [162]) is one of the best general CF parsers available today.

Of course, by reintroducing breadth-first search we are taking chances. The grammar and the input could conspire so that the non-determinism gets hit by each input symbol and our parser will again have exponential time dependency. In practice, however, they never do so and such parsers are very useful.

3.5.8 Conclusion

Figure 3.11 summarizes parsing techniques as they are treated in this book. Nijholt [154] paints a more abstract view of the parsing landscape, based on left-corner parsing. See Deussen [22] for an even more abstracted overview. An early systematic survey was given by Griffiths and Petrick [9].

3.6 The "Strength" of a Parsing Technique

Formally a parsing technique T_1 is stronger (more powerful) than a parsing technique T_2 if T_1 can handle all grammars T_2 can handle but not the other way around. Informally one calls one parsing technique stronger than another if to the speaker it appears to handle a larger set of grammars. Formally this is of course nonsense, since all parsing techniques can handle infinite sets of grammars, and the notion of "larger" is moot. Also, a user grammar designed without explicit aim at a particular parsing technique has an almost zero chance of being amenable to any existing technique anyway. What counts from a user point of view is the effort required to modify the "average" practical grammar so it can be handled by method T, and to undo the damage this modification causes to the parse tree. The *strength* (*power*) of parsing technique T is inversely proportional to that effort.

Almost invariably a strong parser is more complicated and takes more effort to write than a weak one. But since a parser or parser generator (see Section 17.2) needs to be written only once and then can be used as often as needed, a strong parser saves effort in the long run.

Although the notion of a "strong" parser is intuitively clear, confusion can arise when it is applied to the combination of parser and grammar. The stronger the parser is, the fewer restrictions the grammars need to obey and the "weaker" they can afford to be. Usually methods are named after the grammar and it is here where the

	Top-down	Bottom-up
Non-directional methods	Unger parser	CYK parser
Directional Methods, depth-first or breadth-first	The predict/match automaton recursive descent DCG (Definite Clause Grammars) cancellation parsing	The shift/reduce automaton Breadth-first, top-down restricted (Earley) chart parsing
Deterministic directional methods: breadth-first search, with breadth restricted to 1, bounded look-ahead	LL(k)	precedence bounded-right-context LR(k) LALR(k) SLR(k)
unbounded look-ahead	LL-regular	DR (Discriminating-Reverse) LR-regular LAR(m)
Deterministic with postponed ("non-canonical") node identification	LC(k) deterministic cancellation Partitioned LL(k)	total precedence NSLR(k) LR(k,∞) Partitioned LR(k)
Generalized deterministic: maximally restricted breadth-first search	Generalized LL Generalized cancellation Generalized LC	Generalized LR

Fig. 3.11. An overview of context-free parsing techniques

confusion starts. A "strong LL(1) grammar" is more restricted than an "LL(1) grammar"; one can also say that it is more strongly LL(1). The parser for such grammars is simpler than one for (full) LL(1) grammars, and is — can afford to be — weaker. So actually a strong-LL(1) parser is weaker than an LL(1) parser. We have tried to consistently use the hyphen between "strong" and "LL(1)" to show that "strong" applies to "LL(1)" and not to "parser", but not all publications follow that convention, and the reader must be aware. The reverse occurs with "weak-precedence parsers" which are stronger than "precedence parsers" (although in that case there are other differences as well).

3.7 Representations of Parse Trees

The purpose of parsing is to obtain one or more parse trees, but many parsing techniques do not tell you in advance if there will be zero, one, several or even infinitely many parse trees, so it is a little difficult to prepare for the incoming answers. There

are two things we want to avoid: being under-prepared and miss parse trees, and being over-prepared and pre-allocate an excessive amount of memory. Not much published research has gone into this problem, but the techniques encountered in the literature can be grouped into two models: the producer-consumer model and the data structure model.

3.7.1 Parse Trees in the Producer-Consumer Model

In the producer-consumer model the parser is the producer and the program using the parse trees is the consumer. As in all producer-consumer situations in computer science, the immediate question is, which is the main program and which is the subroutine.

The most esthetically pleasing answer is to have them both as equal partners, which can be done using *coroutine*s. Coroutines are explained in some books on principles of programming languages and programming techniques, for example *Advanced Programming Language Design* by R.A. Finkel (Addison-Wesley). There are also good explanations on the Internet.

In the coroutine model, the request for a new parse tree by the user and the offer of a parse tree by the parser are paired up automatically by the coroutine mechanism. The problem with coroutines is that they must be built into the programming language, and no major programming language features them. So coroutines are not a practical solution to parse tree representation.

The coroutine's modern manifestation, the thread, in which the pairing up is done by the operating system or by a light-weight version of it inside the program, is available in some major languages, but introduces the notion of parallelism which is not inherently present in parsing. The UNIX pipe has similar communication properties but is even more alien to the parsing problem.

Usually the parser is the main program and the consumer is the subroutine. Each time the parser has finished constructing a parse tree, it calls the consumer routine with a pointer to the tree as a parameter. The consumer can then decide what to do with this tree: reject it, accept it, store it for future comparison, etc. In this setup the parser can just happily produce trees, but the consumer will probably have to save state data between being called, to be able to choose between parse trees. This is the usual setup in compiler design, where there is only one parse tree and the user state saving is less of a problem.

It is also possible to have the user as the main program, but this places a heavy burden on the parser, which is now forced to keep all state data of its half-finished parsing process when delivering a parse tree. Since data on the stack cannot be saved as state data (except by draconian means) this setup is feasible only with parsing methods that do not use a stack.

With any of these setups the user still has two problems in the general case. First, when the parser produces more than one parse tree the user receives them as separate trees and may have to do considerable comparison to find the differences on which to base further decisions. Second, if the grammar is infinitely ambiguous and the parser produces infinitely many parse trees, the process does not terminate.

So the producer-consumer model is satisfactory for unambiguous grammars, but is problematic for the general case.

3.7.2 Parse Trees in the Data Structure Model

In the data structure model the parser constructs a single data structure which represents all parse trees simultaneously. Surprisingly, this can be done even for infinitely ambiguous grammars; and what is more, it can be done in a space whose size is at most proportional to the third power of the length of the input string. One says that the data structure has *cubic space dependency*.

There are two such representations: parse forests and parse-forest grammars. Although the two are fundamentally the same, they are very different conceptually and practically, and it is useful to treat them as separate entities.

3.7.3 Parse Forests

Since a forest is just a collection of trees, the naive form of a *parse forest* consists of a single node from which all trees in the parse forest are directly reachable. The two parse trees from Figure 3.2 then combine into the parse forest from Figure 3.12, where the numbers in the nodes refer to rule numbers in the grammar of Figure 3.1.

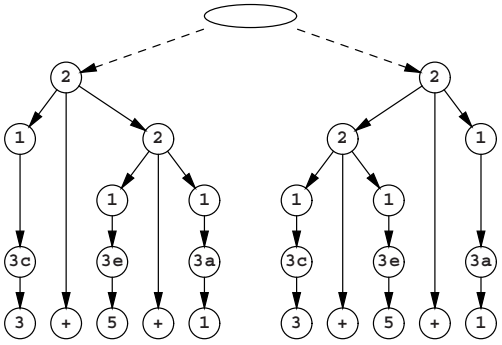


Fig. 3.12. Naive parse forest from the trees in Figure 3.2

When we look at this drawing, we notice two things: the meaning of the dashed arrows differs from that of the drawn arrows; and the resulting tree contains a lot of duplicate subtrees. One also wonders what the label in the empty top node should be.

The meaning of the dashed arrow is "or-or": the empty top node points to *either* the left node marked 2 *or* to the right node, whereas the drawn arrows mean "and-and": the left node marked 2 consists of a node marked **`Sum`** *and* a node marked **`+`** *and* a node marked **`Sum`**. More in particular, the empty top node, which should be labeled **`Sum`** points to two applications of rule 2, each of which produces **`Sum + Sum`**; the leftmost **`Sum`** points to an application of rule 1, the second **`Sum`** points to an

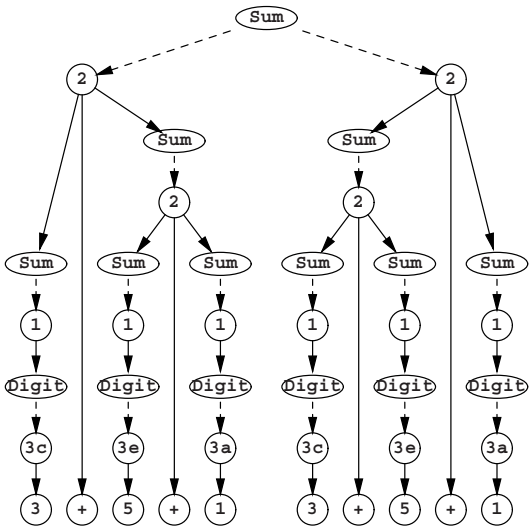


Fig. 3.13. The naive parse forest as an AND-OR tree

application of rule 2, etc. The whole *AND-OR tree* is presented in Figure 3.13, where we see an alteration of nodes labeled with non-terminals, the *OR-nodes* and nodes labeled with rule numbers, the *AND-nodes*. An OR-node for a non-terminal *A* has the rule numbers of the alternatives for *A* as children; an AND-node for a rule number has the components of the right-hand side of the rule as children.

3.7.3.1 Combining Duplicate Subtrees

We are now in a position to combine all the duplicate subtrees in the forest. We do this by having only one copy of a node labeled with a non-terminal *A* and spanning a given substring of the input. If *A* produces that substring in more than one way, more than one or-arrow will emanate from the OR-node labeled *A*, each pointing to an AND-node labeled with a rule number. In this way the AND-OR tree turns into a directed acyclic graph, a dag, which by rights should be called a *parse dag*, although the term "parse forest" is much more usual. The result of our example is shown in Figure 3.14.

It is important to note that two OR-nodes (which represent right-hand sides of rules) can only be combined if all members of the one node are the same as the corresponding members of the other node. It would not do to combine the two nodes marked 2 in Figure 3.14 right under the top; although they both read **`Sum+Sum`**, the **`Sum`**s and even the **`+`**s are not the same. If they were combined, the parse forest would represent more parse trees than correspond with the input; see Problem 3.8.

It is possible to do the combining of duplicate subtrees *during* parsing rather than afterwards, when all trees have been generated. This is of course more efficient, and has the additional advantage that it allows infinitely ambiguous parsings to be

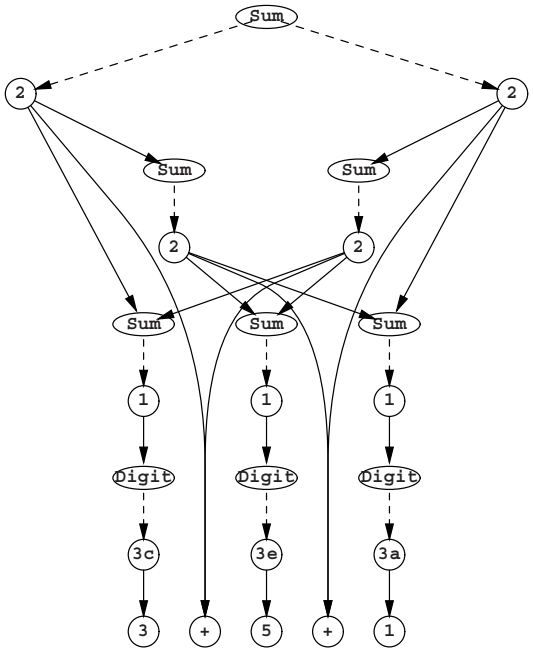


Fig. 3.14. The parse trees of Figure 3.2 as a parse forest

represented in a finite data structure. The resulting parse forest then contains loops (cycles), and is actually a *parse graph*.

Figure 3.15 summarizes the situation for the various Chomsky grammar types in relation to ambiguity. Note that finite-state and context-sensitive grammars cannot

	Most complicated data structure		
Grammar type	unambiguous	ambiguous	infinitely ambiguous
PS	dag	dag	graph
CS	dag	dag	—
CF	tree	dag	graph
FS	list	dag	—

Fig. 3.15. The data structures obtained when parsing with the Chomsky grammar types

be infinitely ambiguous because they cannot contain nullable rules. See Figure 2.16 for a similar summary of production data structures.

3.7.3.2 Retrieving Parse Trees from a Parse Forest

The receiver of a parse forest has several options. For example, a sequence of parse trees can be generated from it, or, perhaps more likely, the data structure can be pruned to eliminate parse trees on various grounds.

Generating parse trees from the parse forest is basically simple: each combination of choices for the or-or arrows is a parse tree. The implementation could be top-down and will be sketched here briefly. We do a depth-first visit of the graph, and for each OR-node we turn one of the outgoing dashed arrows into a solid arrow; we record each of these choices in a backtrack chain. When we have finished our depth-first visit we have fixed one parse tree. When we are done with this tree, we examine the most recent choice point, as provided by the last element of the backtrack chain, and make a different choice there, if available; otherwise we backtrack one step more, etc. When we have exhausted the entire backtrack chain we know we have generated all parse trees. One actualization of a parse tree is shown in Figure 3.16.

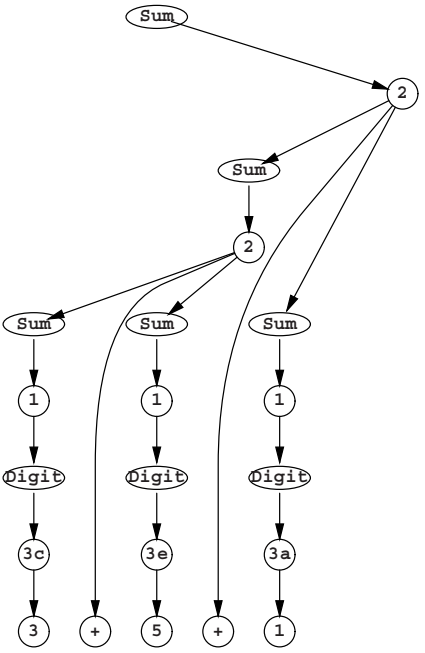


Fig. 3.16. A tree identified in the parse forest of Figure 3.14

It is usually more profitable to first prune the parse forest. How this is done depends on the pruning criteria, but the general technique is as follows. Information is attached to each node in the parse forest, in a way similar to that in attribute grammars (see Section 2.11.1). Whenever the information in a node is contradictory to the criteria for that type of node, the node is removed from the parse forest. This will often make other nodes inaccessible from the top, and these can then be removed as well.

Useful pruning of the parse forest of Figure 3.14 could be based on the fact that the `+` operator is left-associative, which means that `a+b+c` should be parsed as `((a+b)+c)` rather than as `(a+(b+c))`. The criterion would then be that for each

node that has a **+** operator, its right operand cannot be a non-terminal that has a node with a **+** operator. We see that the top-left node marked 2 in Figure 3.14 violates this criterion: it has a **+** operator (in position 2) and a right operand which is a non-terminal (**Sum**) which has a node which has a node (2) which has a **+** operator (in position 4). So this node can be removed, and so can two further nodes. Again the parse tree from Figure 3.16 remains.

The above criterion is a (very) special case of an operator-precedence criterion for arithmetic expressions; see Problem 3.10 for a more general one.

3.7.4 Parse-Forest Grammars

Representing the result of parsing as a *grammar* may seem weird, far-fetched and even somewhat disappointing; after all, should one start with a grammar and a string and do all the work of parsing, just to end up with another grammar? We will, however, see that parse-forest grammars have quite a number of advantages. But none of these advantages is immediately obvious, which is probably why parse-forest grammars were not described in the literature until the late 1980s, when they were introduced by Lang [210, 220, 31], and by Billot and Lang [164]. The term "parse-forest grammar" seems to be used first by van Noord [221].

Figure 3.17 presents the parse trees of Figure 3.2 as a parse-forest grammar, and it is interesting to see how it does that. For every non-terminal *A* in the original

```
    Sum_S  ->  Sum_1_5
  Sum_1_5  ->  Sum_1_1 + Sum_3_3
  Sum_1_5  ->  Sum_1_3 + Sum_5_1
  Sum_1_3  ->  Sum_1_1 + Sum_3_1
  Sum_3_3  ->  Sum_3_1 + Sum_5_1
  Sum_1_1  ->  Digit_1_1
Digit_1_1  ->  3
  Sum_3_1  ->  Digit_3_1
Digit_3_1  ->  5
  Sum_5_1  ->  Digit_5_1
Digit_5_1  ->  1
```

Fig. 3.17. The parse trees of Figure 3.2 as a parse-forest grammar

grammar that produces an input segment of length *l* starting at position *i*, there is a non-terminal *A_i_l* in the parse-forest grammar, with rules that show how *A_i_l* produces that segment. For example, the existence of **Sum_1_5** in the parse-tree grammar shows that **Sum** produces the whole input string (starting at position 1, with length 5); the fact that there is more than one rule for **Sum_1_5** shows that the parsing was ambiguous; and the two rules show the two possible ways **Sum_1_5** produces the whole input string. When we use this grammar to generate strings, it generates just the input sentence **3+5+1**, but is generates it twice, in accordance with the ambiguity.

We write *A_i_l* rather than $A_{i,l}$ because *A_i_l* represents the name of a grammar symbol, not a subscripted element of an entity *A*: there is no table or matrix *A*. Nor is there any relationship between *A_i_l* and say *A_i_m*: each *A_i_l* is a separate name of a grammar symbol.

Now for the advantages. First, parse-forest grammars implement in a graphical way the concept, already expressed less directly in the previous section, that there should be exactly one entity that describes how a given non-terminal produces a given substring of the input.

Second, it is mathematically beautiful: parsing a string can now be viewed as a function which maps a grammar onto a more specific grammar or an error value. Rather than three concepts — grammars, input strings, and parse forests — we now need only two: grammars and input strings. More practically, all software used in handling the original grammar is also available for application to the parse-forest grammar.

Third, parse-forest grammars are easy to clean up after pruning, using the algorithms from Section 2.9.5. For example, applying the disambiguation criterion used in the previous section to the rules of the grammar in Figure 3.17 identifies the first rule for `Sum_1_5` as being in violation. Removing this rule and applying the grammar clean-up algorithm yields the unambiguous grammar of Figure 3.18, which corresponds to the tree in Figure 3.16.

```
    Sum_S  ->  Sum_1_5
  Sum_1_5  ->  Sum_1_3 + Sum_5_1
  Sum_1_3  ->  Sum_1_1 + Sum_3_1
  Sum_1_1  ->  Digit_1_1
Digit_1_1  ->  3
  Sum_3_1  ->  Digit_3_1
Digit_3_1  ->  5
  Sum_5_1  ->  Digit_5_1
Digit_5_1  ->  1
```

Fig. 3.18. The disambiguated and cleaned-up parse-forest grammar for Figure 3.2

Fourth, representing infinitely ambiguous parsings is trivial: the parse-forest grammar just produces infinitely many (identical) strings. And producing infinitely many strings is exactly what grammars normally do.

And last but probably not least, it fits in very well with the interpretation of parsing as intersection, an emerging and promising approach, further discussed in Chapter 13.

Now it could be argued that parse forests and parse-forest grammars are actually the same and that the pointers in the first have just been replaced by names in the second, but that would not be fair. Names are more powerful than pointers, since a pointer can point only to one object, whereas a name can identify several objects, through overloading or non-determinism: names are multi-way pointers. More in particular, the name `Sum_1_5` in Figure 3.17 identifies *two* rules, thus playing the

role of the top OR-node in Figure 3.14. We see that in parse-forest grammars we get the AND-OR tree mechanism free of charge, since it is built into the production mechanism of grammars.

3.8 When are we done Parsing?

Since non-directional parsers process the entire input at once and summarize it into a single data structure, from which parse trees can then be extracted, the question of when the parsing is done does not really arise. The first stage is done when the data structure is finished; extracting the parse trees is done when they are exhausted or the user is satisfied.

In principle, a directional parser is finished when it is in an accepting state and the entire input has been consumed. But this is a double criterion, and sometimes one of these conditions implies the other; also other considerations often play a role. As a result, for directional parsers the question has a complex answer, depending on a number of factors:

- Is the parser at the end of the input? That is, has it processed completely the last token of the input?
- Is the parser in an accepting state?
- Can the parser continue, i.e, is there a next token and can the parser process it?
- Is the parser used to produce a parse tree or is just recognition enough? In the first case several situations can arise; in the second case we just get a yes/no answer.
- If we want parsings, do we want them all or is one parsing enough?
- Does the parser have to accept the entire input or is it used to isolate a prefix of the input that conforms to the grammar? (A string x is a *prefix* of a string y if y begins with x.)

The answers to the question whether we have finished the parsing are combined in the following table, where EOI stands for "end of input" and the yes/no answer for recognition is supplied between parentheses.

at end of input?	can continue?	in an accepting state? yes	in an accepting state? no
yes	yes	prefix identified / continue	continue
yes	no	OK (yes)	premature EOI (no)
no	yes	prefix identified / continue	continue
no	no	prefix identified & trailing text (no)	error in input (no)

Some answers are intuitively reasonable: if the parser can continue in a non-accepting state, it should do so; if the parser cannot continue in a non-accepting state, there was an error in the input; and if the parser is in an accepting state at the end of the input and cannot continue, parsing was successful. But others are more

complicated: if the parser is in an accepting state, we have isolated a prefix, even if the parser could continue and/or is at EOI. If that is what we want we can stop, but usually we want to continue if we can: with the grammar `S→a | ab` and the input `ab` we could stop after the `a` and declare the `a` a prefix, but it is very likely we want to continue and get the whole `ab` parsed. This could be true even if we are at EOI: with the grammar `S→a | aB` where `B` produces ε and the input `a` we need to continue and parse the `B`, if we want to obtain all parsings. And if the parser cannot, we have recognized a string in the language with what error messages usually call "trailing garbage".

Note that "premature EOI" (the input is a prefix of a string in the language) is the dual of "prefix isolated" (a prefix of the input is a string in the language). If we are looking for a prefix we usually want the longest possible prefix. This can be implemented by recording the most recent position *P* in which a prefix was recognized and continuing parsing until we get stuck, at the end of the input or at an error. *P* is then the end of the longest prefix.

Many directional parsers use look-ahead, which means that there must always be enough tokens for the look-ahead, even at the end of the input. This is implemented by introducing an *end-of-input* token, for example `#` or any other token that does not occur elsewhere in the grammar. For a parser that uses *k* tokens of look-ahead, *k* copies of `#` are appended to the input string; the look-ahead mechanism of the parser is modified accordingly; see for example Section 9.6. The only accepting state is then the state in which the first `#` is about to be accepted, and it always indicates that the parsing is finished.

This simplifies the situation and the above table considerably since now the parser cannot be in an accepting state when not at the end of the input. This eliminates the two prefix answers from the table above. We can then superimpose the top half of the table on the bottom half, after which the leftmost column becomes redundant. This results in the following table:

can continue?	in an accepting state? yes	no
yes	—	continue
no	OK (yes)	error in input / premature EOI (no)

where we leave the check to distinguish between "error in input" and "premature EOI" to the error reporting mechanism.

Since there is no clear-cut general criterion for termination in directional parsers, each parser comes with its own stopping criterion, a somewhat undesirable state of affairs. In this book we will use end-of-input markers whenever it is helpful for termination, and, of course, for parsers that use look-ahead.

3.9 Transitive Closure

Many algorithms in parsing (and in other branches of computer science) have the property that they start with some initial information and then continue to draw conclusions from it based on some inference rules until no more conclusions can be drawn. We have seen two examples already, with their inference rules, in Sections 2.9.5.1 and 2.9.5.2. These inference rules were quite different, and in general inference rules can be arbitrarily complex. To get a clear look at the algorithm for drawing conclusions, the closure algorithm, we shall now consider one of the simplest possible inference rules: transitivity. Such rules have the form

if $A \otimes B$ and $B \otimes C$ then $A \otimes C$

where $\otimes$ is any operator that obeys the rule. The most obvious one is $=$, but $<$, $\leq$ and many others also do. But note that, for example, $\neq$ (not equal) does not.

As an example we shall consider the computation of the "left-corner set" of a non-terminal. A non-terminal B is in the *left-corner* set of a non-terminal A if there is a derivation $A \xrightarrow{*} B \cdots$; it is sometimes useful to know this, because among other things it means that A can begin with anything B can begin with.

Given the grammar

```
S_S  →  S T
 S   →  A a
 T   →  A t
 A   →  B b
 B   →  C c
 C   →  x
```

how can we find out that **C** is in the left-corner set of **S**? The rules **S→ST** and **S→Aa** in the grammar tell us immediately that **S** and **A** are in the left-corner set of **S**. We write this as **S∠S** and **A∠S**, where ∠ symbolizes the left corner. It also tells us **A∠T**, **B∠A**, and **C∠B**. This is our initial information (Figure 3.19(*a*)).

(*a*)	(*b*)	(*c*)	(*d*)
S∠S	S∠S ✔	B∠S ✔	C∠S ✔
A∠S	A∠S ✔	C∠S	
A∠T	A∠S ✔	C∠T	
B∠A	B∠S	C∠S ✔	
C∠B	B∠T	C∠T ✔	
	B∠S ✔		
	B∠T ✔		
	C∠A		
	C∠A ✔		

Fig. 3.19. The results of the naive transitive closure algorithm

Now it is easy to see that if A is in the left-corner set of B and B is in the left-corner set of C, then A is also in the left-corner of C. In a formula:

$$A\angle B \wedge B\angle C \Rightarrow A\angle C$$

This is our *inference rule*, and we will use it for drawing new conclusions, or "inferences", by pairwise combining known facts to produce more known facts. The *transitive closure* is then obtained by applying the inference rules until no more new facts are produced. The facts are also called "relations" in the transitive-closure context, although formally $\angle$ is the (binary) relation, and $A\angle B$ and $B\angle C$ are "instances" of that relation.

Going through the list in Figure 3.19(a) we first combine **S**$\angle$**S** and **S**$\angle$**S**. This yields **S**$\angle$**S**, which is rather disappointing since we knew that already; it is in Figure 3.19(b), marked with a ✔ to show that it is not new. The combination (**S**$\angle$**S**, **A**$\angle$**S**) yields **A**$\angle$**S**, but we already knew that too. No other facts combine with **S**$\angle$**S**, so we continue with **A**$\angle$**S**, which yields **A**$\angle$**S** and **B**$\angle$**S**; the first is old, the second our first new discovery. Then (**A**$\angle$**T**, **B**$\angle$**A**) yields **B**$\angle$**T**, etc., and the rest of the results of the first round can be seen in Figure 3.19(b).

The second round combines the three new facts with the old and new ones. The first new discovery is **C**$\angle$**S** from **A**$\angle$**S** and **C**$\angle$**A** (c); **C**$\angle$**T** follows.

The third round combines the two new facts in (c) with those in (a), (b), and (c), but finds no new facts; so the algorithm terminates with 10 facts.

Note that we have already implemented an optimization in this naive algorithm: the basic algorithm would start the second and subsequent rounds by pairing up *all* known facts with all known facts, rather than just the new ones.

It is often useful to represent the facts or relations in a graph, in which they are arcs. The initial situation is shown in Figure 3.20(a), the final one in (b). The numbers

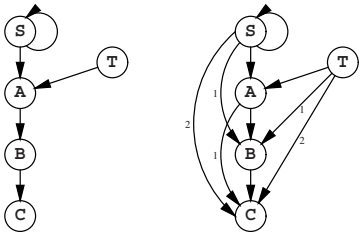


Fig. 3.20. The left-corner relation as a graph

next to the arrows indicate the rounds in which they were added.

The efficiency of the closure algorithm of course depends greatly on the inference rule it uses, but the case for the transitive rule has been studied extensively. There are three main ways to do transitive closure: naive, traditional, and advanced; we will discuss each of them briefly. The naive algorithm, sketched above, is usually quite efficient in normal cases but may require a large number of rounds to converge in exceptional cases on very large graphs. Also it recomputes old results several times, as we see in Figure 3.19: of the 15 results 10 were old. But given the size of "normal" grammars, the naive algorithm is satisfactory in almost all situations in parsing.

The traditional method to do transitive closure is to use Warshall's algorithm [409]. It has the advantage that it is very simple to implement and that the time it requires depends only on the number of nodes N in the graph and not on the number of arcs, but it has the disadvantage that it always requires $O(N^3)$ time. It always loses in any comparison to any other closure algorithm.

The advanced algorithms avoid the inefficiencies of the naive algorithm: 1. cycles in the graph are contracted as "strongly connected components"; 2. the arcs are combined in an order which avoids duplicate conclusions and allows sets of arcs to be copied rather than recomputed; 3. efficient data representations are used. For example, an advanced algorithm would first compute all outgoing arcs at `A` and then copy them to `T` rather than recomputing them for `T`. The first advanced transitive closure algorithm was described by Tarjan [334]. They are covered extensively in many other publications; see Nuutila [412] and the Internet. They require time proportional to the number of conclusions they draw.

Advanced transitive closure algorithms are very useful in large applications (databases, etc.) but their place in parsing is doubtful. Some authors recommend their use in LALR parser generators but the grammars used would have to be very large for the algorithmic complexity to pay off.

The advantage of emphasizing the closure nature of algorithms is that one can concentrate on the inference rules and take the underlying closure algorithm for granted; this can be a great help in designing algorithms. Most algorithms in parsing are, however, simple enough as to not require decomposition into inference rules and closure for their explanation. We will therefore use inference rules only where they are helpful in understanding (Section 9.7.1.3) and where they are part of the culture (Section 7.3, chart parsing). For the rest we will present the algorithms in narrative form, and point out in passing that they are transitive-closure algorithms.

3.10 The Relation between Parsing and Boolean Matrix Multiplication

There is a remarkable and somewhat mysterious relationship between parsing and Boolean matrix multiplication, in that it is possible to turn one into the other and vice versa, with a lot of ifs and buts. This has interesting implications.

A Boolean matrix is a matrix in which all entries are either 0 or 1. If the indexes of a matrix T represent towns, the element $T_{i,j}$ could, for example, indicate the existence of a direct railroad connection from town i to town j. Such a matrix can be multiplied by another Boolean matrix $U_{j,k}$, which could, for example, indicate the existence of a direct bus connection from town j to town k. The result $V_{i,k}$ (the product of T and U) is a Boolean matrix which indicates if there is a connection from town i to town k by first using a train and then a bus. This immediately shows how $V_{i,k}$ must be computed: it should have a 1 if there is a j for which both $T_{i,j}$ and $U_{j,k}$ hold a 1, and a 0 otherwise. In a formula:

$$V_{i,k} = (T_{i,1} \wedge U_{1,k}) \vee (T_{i,2} \wedge U_{2,k}) \vee \cdots \vee (T_{i,n} \wedge U_{n,k})$$

where $\wedge$ is the Boolean AND, $\vee$ is the Boolean OR, and n is the size of the matrices. This means that $O(n)$ actions are required for each entry in V, of which there are n^2; so the time dependency of this algorithm is $O(n^3)$.

Figure 3.21 shows an example; the boxed row $T_{2,*}$ combines with the boxed column $U_{*,2}$ to produce the boxed entry $V_{2,2}$. Boolean matrix multiplication is not

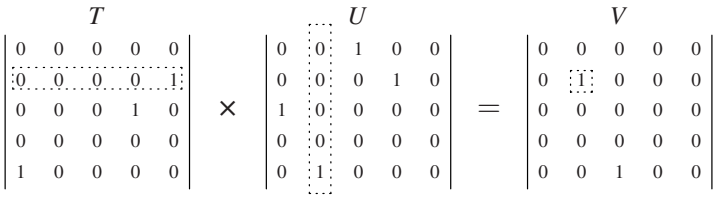


Fig. 3.21. Boolean matrix multiplication

commutative: it is quite possible that there is a train-bus connection but no bus-train connection from one town to another, so $T \times U$ will in general not be equal to $U \times T$. Note also that this differs from transitive closure: in transitive closure a single relation is followed an unbounded number of times, whereas in Boolean matrix multiplication first one relation is followed and then a second.

The above is a trivial application of *Boolean matrix multiplication* (*BMM*), but BMM is very important in many branches of mathematics and industry, and there is a complete science on how to perform it efficiently.[3] Decades of concentrated effort have resulted in a series of increasingly more efficient and complicated algorithms. V. Strassen[4] was the first to break the $O(n^3)$ barrier with an $O(n^{2.81\cdots})$ algorithm, and the present record stands at $O(n^{2.376\cdots})$; it dates from 1987. It is clear that at least $O(n^2)$ actions are required, but it is unlikely that that efficiency can be achieved.

More important from our viewpoint is the fact that in 1975 Valiant [18] showed how a CF parsing problem can be converted into a BMM problem. In particular, if you can multiply two Boolean matrices of size $n \times n$ in $O(n^k)$ actions, you can parse a string of length n in $O(n^k) + O(n^2)$ actions, where the $O(n^2)$ is the cost of the conversion. So we can do general CF parsing in $O(n^{2.376\cdots})$, which is indeed better than the cubic time dependency of the CYK algorithm. But the actions of both Valiant's algorithm and the fast BMM are extremely complicated and time-consuming, so this approach would only be better for inputs of millions of symbols or more. On top of that it requires all these symbols to be in memory, as it is a non-directional method, and the size of the data structures it uses is $O(n^2)$, which means that it can only be run profitably on a machine with terabytes of main memory. In short, its significance is theoretical only.

[3] For a survey see V. Strassen, "Algebraic complexity theory", *in* Handbook of Theoretical Computer Science, vol. A, Jan van Leeuwen, Ed. Elsevier Science Publishers, Amsterdam, The Netherlands, pp. 633-672, 1990.

[4] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, 13:354-356, 1969.

In 2002 Lee [39] showed how a BMM problem can be converted into a CF parsing problem. More in particular, if you can do general CF parsing of a string of length n in $O(n^{3-\delta})$ actions, you can multiply two Boolean matrices of size $n \times n$ in $O(n^{3-\delta/3})$ actions. There is again a conversion cost of $O(n^2)$, but since δ can be at most 2 (in which unlikely case parsing could be done in $O(n)$), $O(n^{3-\delta/3})$ is at least $O(n^{2\frac{1}{3}})$, which dominates the $O(n^2)$; note that for $\delta = 0$ the usual $O(n^3)$ bounds for both problems result. The computational efforts involved in Lee's conversion are much smaller than those in Valiant's technique, so a really fast general CF parsing algorithm would likely supply a fast practical BMM algorithm. Such a fast general CF parsing algorithm would have to be non-BMM-dependent and have a time complexity better than $O(n^3)$; unfortunately no such algorithm is known.

General CF parsing and Boolean matrix multiplication have in common that the efficiencies of the best algorithms for them are unknown. Figure 3.22 summarizes the possibilities. The horizontal axis plots the efficiency of the best possible general

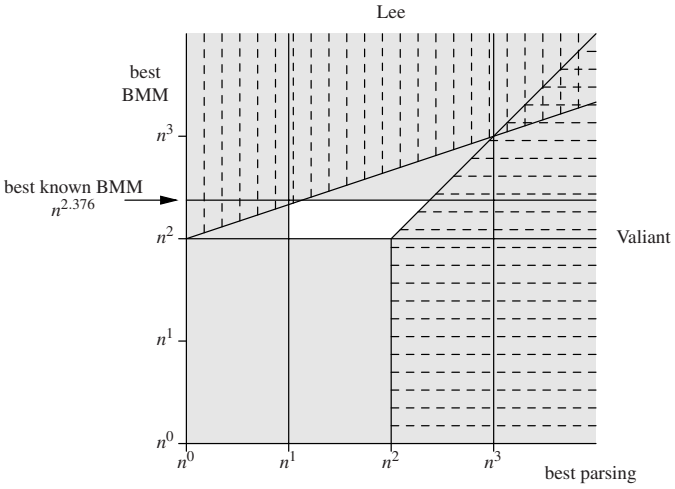


Fig. 3.22. Map of the best parser versus best BMM terrain

CF parsing algorithm; the vertical axis plots the efficiency of the best possible BMM algorithm. A position in the graph represents a combination of these values. Since these values are unknown, we do not know which point in the graph corresponds to reality, but we can exclude several areas.

The grey areas are excluded on the grounds of existing algorithms. For example, the grey area on the right of the vertical line at n^3 is excluded because we have the CYK algorithm, which does general CF parsing in $O(n^3)$; so the pair (best parser, best BMM) cannot have a first component which is larger than $O(n^3)$. Likewise the area left of the vertical line at n^1 represents parsing algorithms that work in less than $O(n)$, which is impossible since the parser must touch each token. BMM requires

at least $O(n^2)$ actions, but an algorithm for $O(n^{2.376\cdots})$ is available; this yields two horizontal forbidden areas.

The shading marks the areas that are excluded by the Valiant and Lee conversion algorithms. Valiant's result excludes the horizontally shaded area on the right; Lee's result excludes the vertically shaded area at the top. The combination of the efficiencies of the true best parsing and BMM algorithms can only be situated in the white unshaded area in the middle.

Extensive research on the BMM problem has not yielded a usable algorithm that is substantially better than $O(n^3)$; since BMM can be converted to parsing this could explain why the admittedly less extensive research on general CF parsing has not yielded a better than $O(n^3)$ algorithm, except through BMM. On the other hand Figure 3.22 shows that it is still possible for general CF parsing to be linear ($O(n^1)$) and BMM to be worse than $O(n^2)$.

Rytter [34] has linked general CF parsing to a specific form of shortest-path computation in a lattice, with comparable implications.

Greibach [389] describes the "hardest context-free language", a language such that if we can parse it in time $O(n^x)$, we can parse any CF language in time $O(n^x)$. Needless to say, it's hard to parse. The paper implicitly uses a parsing technique which has received little attention; see Problem 3.7.

3.11 Conclusion

Grammars allow sentences to be produced through a well-defined process, and the details of this process determines the structure of the sentence. Parsing recovers this structure either by imitating the production process (top-down parsing) or by rolling it back (bottom-up parsing). The real work goes into gathering information to guide the structure recovery process efficiently.

There is a completely different and — surprisingly — grammarless way to do CF parsing, "data-oriented parsing", which is outside the scope of this book. See Bod [348] and the Internet.

Problems

Problem 3.1: Suppose all terminal symbols in a given grammar are different. Is that grammar unambiguous?

Problem 3.2: Write a program that, given a grammar G and a number n, computes the number of different parse trees with n leaves (terminals) G allows.

Problem 3.3: If you are familiar with an existing parser (generator), identify its parser components, as described on page 69.

Problem 3.4: The maze preprocessing algorithm in Section 3.5.4 eliminates all rooms with three walls; rules with two or four walls are acceptable in a deterministic maze. What about rooms with zero or one wall? How do they affect the algorithm and the result? Is it possible/useful to eliminate them too?

Problem 3.5: Construct an example in which a deterministic bottom-up parser will have to perform k actions at position k, for a certain k.

Problem 3.6: *Project:* There are several possible paths through the maze in Figure 3.10(b), so a maze defines a set of paths. It is easy to see that these paths form a regular set. This equates a maze to a regular grammar. Develop this analogy, for example: 1. Derive the regular grammar from some description of the maze. 2. How does the subset algorithm (Section 5.3.1) transform the maze? 3. Is it possible to generate a set of mazes so that together they define a given CF set?

Problem 3.7: *Project:* Study the "translate and cross out matching parentheses" parsing method of Greibach [389].

Problem 3.8: Show that a version of Figure 3.14 in which the nodes marked 2 near the top are combined represents parse trees that are not supported by the input.

Problem 3.9: Implement the backtracking algorithm sketched in Section 3.7.3.

Problem 3.10: Assume arithmetic expressions are parsed with the highly ambiguous grammar

```
Expr_S  ->  Number
 Expr   ->  Expr Operator Expr | ( Expr )
Operator -> + | - | × | / | ↑
```

with an appropriate definition of **`Number`**. Design a criterion that will help prune the resulting parse forest to obtain the parse tree that obeys the usual precedences for the operators. For example, $4+5\times 6+8$ should come out as $((4+(5\times 6))+8)$. Take into account that the first four operators are left-associative, but the exponentiation operator $\uparrow$ is right-associative: $6/6/6$ is $((6/6)/6)$ but $6\uparrow 6\uparrow 6$ is $(6\uparrow(6\uparrow 6))$.

Problem 3.11: *Research project:* Some parsing problems involve extremely large CF grammar, with millions of rules. Such a grammar is generated by program and results from incorporating finite context conditions into the grammar. It is usually very redundant, containing many very similar rules, and very ambiguous. Many general CF parsers are quadratic in the size of the grammar, which for ten million rules brings in a factor of 10^{14}. Can parsing techniques be designed that work well on such grammars? (See also Problem 4.5.)

Problem 3.12: *Extensible Project:* 1. A string S is *balanced* for a token pair (t_1,t_2) if $\#t_1 = \#t_2$ for S and $\#t_1 \geq \#t_2$ for all prefixes of S, where $\#t$ is the number of occurrences of t in S or a prefix of it. A token pair (t_1,t_2) is a *parentheses pair* for a grammar G if all strings in $\mathcal{L}(G)$ are balanced for (t_1,t_2). Design an algorithm to check if a token pair (t_1,t_2) is a parentheses pair for a given grammar G: a) under the simplifying but reasonable assumption that parentheses pairs occur together in the right hand side of a rule (for example, as in **`F->(E)`**), and b) in the general case.

2. A token t_1 in position i in a string *matches* a token t_2 in a position j if the string segment $i+1\cdots j-1$ between them is balanced for (t_1,t_2). A parentheses pair (t_1,t_2) is *compatible with* a parentheses pair (u_1,u_2) if every segment between a t_1 and its matching t_2 in every string in $\mathcal{L}(G)$ is balanced for (u_1,u_2). Show that if (t_1,t_2) is compatible with (u_1,u_2), (u_1,u_2) is compatible with (t_1,t_2).

3. Design an algorithm to find a largest set of compatible parentheses pairs for a given grammar.

4. Use the set of parentheses pairs to structure sentences in $\mathcal{L}(G)$ in linear time.

5. Derive information from G about the segments of strings in $\mathcal{L}(G)$ that are not structured in that process, for example regular expressions.

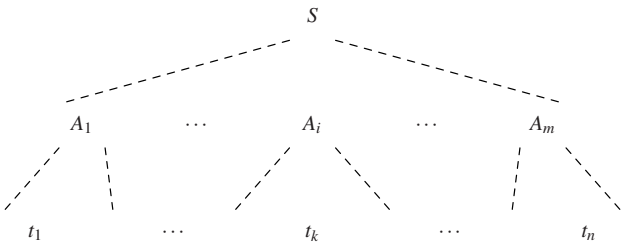
6. Devise further techniques to exploit the parentheses skeleton of CF languages.

4

General Non-Directional Parsing

In this chapter we will present two general parsing methods, both non-directional: Unger's method and the CYK method. These methods are called non-directional because they access the input in a seemingly arbitrary order. They require the entire input to be in memory before parsing can start.

Unger's method is top-down; if the input belongs to the language at hand, it must be derivable from the start symbol of the grammar, say S. Therefore, it must be derivable from a right-hand side of the start symbol, say $A_1 A_2 \cdots A_m$. This, in turn, means that A_1 must derive a first part of the input, A_2 a second part, etc. If the input sentence is $t_1 t_2 \cdots t_n$, this demand can be depicted as follows:



Unger's method tries to find a partition of the input that fits this demand. This is a recursive problem: if a non-terminal A_i is to derive a certain part of the input, there must be a partition of this part that fits a right-hand side of A_i. Ultimately, such a right-hand side must consist of terminal symbols only, and these can easily be matched with the current part of the input.

The CYK method approaches the problem the other way around: it tries to find occurrences of right-hand sides in the input; whenever it finds one, it makes a note that the corresponding left-hand side derives this part of the input. Replacing the occurrence of the right-hand side with the corresponding left-hand side results in some sentential forms that derive the input. These sentential forms are again the

subject of a search for right-hand sides, etc. Ultimately, we may find a sentential form that both derives the input sentence and is a right-hand side of the start symbol.

In the next two sections, these methods are investigated in detail.

4.1 Unger's Parsing Method

The input to Unger's parsing method [12] consists of a CF grammar and an input sentence. We will first discuss Unger's parsing method for grammars without ε-rules and without loops (see Section 2.9.4). Then, the problems introduced by ε-rules will be discussed, and the parsing method will be modified to allow for all CF grammars.

4.1.1 Unger's Method without ε-Rules or Loops

To see how Unger's method solves the parsing problem, let us consider a small example. Suppose we have a grammar rule

$S \rightarrow ABC \mid DE \mid F$

and we want to find out whether S derives the input sentence $pqrs$. The initial parsing problem can then be schematically represented as:

S
$pqrs$

For each right-hand side we must first generate all possible partitions of the input sentence. Generating partitions is not difficult: if we have m cups, numbered from 1 to m, and n marbles, numbered from 1 to n, we have to find all possible partitions such that each cup contains at least one marble, the numbers of the marbles in any cup are consecutive, and any cup does not contain lower-numbered marbles than any marble in a lower-numbered cup. We proceed as follows: first, we put marble 1 in cup 1, and then generate all partitions of the other $n-1$ marbles over the other $m-1$ cups. This gives us all partitions that have marble 1 and only marble 1 in the first cup. Next, we put marbles 1 and 2 in the first cup, and then generate all partitions of the other $n-2$ marbles over the other $m-1$ cups, etc. If n is less than m, no partition is possible.

Partitioning the input corresponds to partitioning the marbles (the input symbols) over the cups (the right-hand side symbols). If a right-hand side has more symbols than the sentence, no partition can be found (there being no ε-rules). For the first right-hand side the following partitions must be tried:

S		
A	B	C
p	q	rs
p	qr	s
pq	r	s

The first partition results in the following sub-problems: does A derive p, does B derive q, and does C derive rs? These sub-problems must all be answered in the affirmative, or the partition is not the right one.

For the second right-hand side, we obtain the following partitions:

S	
D	E
p	qrs
pq	rs
pqr	s

The last right-hand side results in the following partition:

S
F
$pqrs$

All these sub-problems deal with shorter sentences, except the last one. They will all lead to similar split-ups, and in the end many will fail because a terminal symbol in a right-hand side does not match the corresponding part of the partition. The only partition that causes some concern is the last one. It is as complicated as the one we started with. This is the reason that we have disallowed loops in the grammar. If the grammar has loops, we may get the original problem back again and again. For example, if there is a rule $F \rightarrow S$ in the example above, this will certainly happen.

The above demonstrates that we have a search problem here, and we can attack it with either the depth-first or the breadth-first search technique (see Section 3.5.2). Unger uses depth-first search.

In the following discussion, the grammar of Figure 4.1 will serve as an example. This grammar represents the language of simple arithmetic expressions, with opera-

```
Expr_S  ->  Expr + Term | Term
 Term   ->  Term × Factor | Factor
Factor  ->  ( Expr ) | i
```

Fig. 4.1. A grammar describing simple arithmetic expressions

tors `+` and `×`, and operand `i`. We will use the sentence `(i+i)×i` as input example. So the initial problem can be represented as:

`Expr`
`(i+i)×i`

Fitting the first alternative of `Expr` with the input `(i+i)×i` results in a list of 15 partitions, shown in Figure 4.2. We will not examine them all here, although the unoptimized version of the algorithm requires this. We will only examine the partitions that have at least some chance of succeeding: we can eliminate all partitions that do not match the terminal symbol of the right-hand side. So the only partition worth investigating further is:

Expr		
Expr	**+**	**Term**
(i+i)×i		
(i+i)	×i	
(i+i)×	i	
(i+i)	×i	
(i+i)×	i	
(i+i)	×i	
(i+i)×	i	

Fig. 4.2. All partitions for **Expr→Expr+Term**

Expr		
Expr	**+**	**Term**
(i	+	i)×i

The first sub-problem here is to find out whether and, if so, how **Expr** derives **(i**. We cannot partition **(i** into three non-empty parts because it only consists of 2 symbols. Therefore, the only rule that we can apply is the rule **Expr→Term**. Similarly, the only rule that we can apply next is the rule **Term→Factor**. So we now have

Expr
Term
Factor
(i

However, this is impossible, because the first right-hand side of **Factor** has too many symbols, and the second one consists of one terminal symbol only. Therefore, the partition we started with does not fit, and it must be rejected. The other partitions were already rejected, so we can conclude that the rule **Expr→Expr+Term** does not derive the input.

The second right-hand side of **Expr** consists of only one symbol, so we only have one partition here, consisting of one part. Partitioning this part for the first right-hand side of **Term** again results in 15 possibilities, of which again only one has a chance of succeeding:

Expr		
Term		
Term	×	Factor
(i+i)	×	i

Continuing our search, we will find the following derivation (the only one to be found):

```
Expr →
Term →
Term × Factor →
Factor × Factor →
( Expr ) × Factor →
( Expr + Term ) × Factor →
( Term + Term ) × Factor →
( Factor + Term ) × Factor →
( i + Term ) × Factor →
( i + Factor ) × Factor →
( i + i ) × Factor →
( i + i ) × i
```

This example demonstrates several aspects of the method: even small examples require a considerable amount of work, but even some simple checks can result in huge savings. For example, matching the terminal symbols in a right-hand side with the partition at hand often leads to the rejection of the partition without investigating it any further. Unger [12] presents several more of these checks. For example, one can compute the minimum length of strings of terminal symbols derivable from each non-terminal. Once it is known that a certain non-terminal only derives terminal strings of length at least n, all partitions that fit this non-terminal with a substring of length less than n can be immediately rejected.

4.1.2 Unger's Method with ε-Rules

So far, we only have dealt with grammars without ε-rules, and not without reason. Complications arise when the grammar contains ε-rules, as is demonstrated by the following example: consider the grammar rule $S \rightarrow ABC$ and input sentence pqr. If we want to examine whether this rule derives the input sentence, and we allow for ε-rules, many more partitions will have to be investigated, because each of the non-terminals A, B, and C may derive the empty string. In this case, generating all partitions proceeds just as above, except that we first generate the partitions that have no marble at all in the first cup, then the partitions that have marble 1 in the first cup, etc.:

S		
A	B	C
		pqr
	p	qr
	pq	r
	pqr	
p		qr
p	q	r
p	qr	
pq		r
pq	r	
pqr		

Now suppose that we are investigating whether B derives pqr, and suppose there is a rule $B \rightarrow SD$. Then, we will have to investigate the following partitions:

B	
S	D
	pqr
p	qr
pq	r
pqr	

It is the last of these partitions that will cause trouble: in the process of finding out whether S derives pqr, we end up asking the same question again, in a different context. If we are not careful and do not detect this, our parser will loop forever, or run out of memory.

When searching along this path, we are looking for a derivation that is using a recursive loop in the grammar of the form $S \rightarrow \cdots \rightarrow \alpha S \beta$. If the grammar contains ε-rules and the parser must assume that α and β can produce ε, this loop will cause the parser to ask the question "does S derive pqr?" over and over again.

If α and β can indeed produce ε, there are infinitely many derivations to be found along this path, provided that there is at least one, so we will never be able to present them all. The only interesting derivations are the ones without the loop. Therefore, we will cut off the search process in these cases. On the other hand, if α and β cannot both produce ε, a cut-off will not do any harm either, because a second search along this path is doomed to fail anyway, if the initial search did not succeed.

So we can avoid the problem altogether by cutting off the search process in these cases. Fortunately, this is not a difficult task. All we have to do is to maintain a list of questions that we are currently investigating. Before starting to investigate a new question (for example "does S derive pqr?") we first check that the question does not already appear in the list. If it does, we do not investigate this question. Instead, we proceed as if the question were answered negatively.

Consider for example the following grammar:

```
S  →  LSD | ε
L  →  ε
D  →  d
```

This grammar generates sequences of **d**s in an awkward way. The complete search for the questions **S** $\stackrel{*}{\rightarrow}$ **d**? and **S** $\stackrel{*}{\rightarrow}$ **dd**? is depicted in Figure 4.3. Figure 4.3

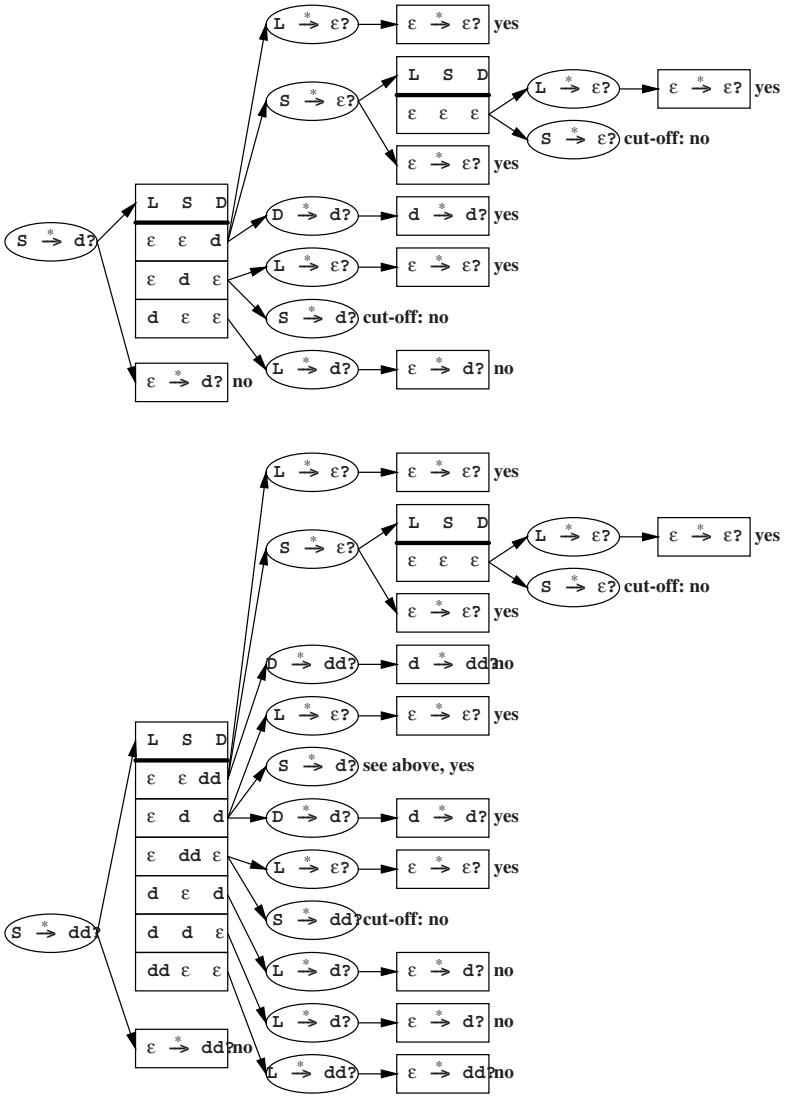

S → d?
L S D
ε ε d
ε d ε
d ε ε
ε → d? no
L → ε? ε → ε? yes
S → ε? L S D / ε ε ε
L → ε? ε → ε? yes
S → ε? cut-off: no
ε → ε? yes
D → d? d → d? yes
L → ε? ε → ε? yes
S → d? cut-off: no
L → d? ε → d? no
S → dd?
L S D
ε ε dd
ε d d
ε dd ε
d ε d
d d ε
dd ε ε
ε → dd? no
D → dd? d → dd? no
S → d? see above, yes
L → d? ε → d? no

Fig. 4.3. Unger's parser at work for the sentences **d** and **dd**

must be read from left to right, and from top to bottom. The questions are drawn

in an ellipse, with the split-ups over the right-hand sides in boxes. A question is answered affirmatively if at least one of the boxes results in a "yes". In contrast, a partition only results in an affirmative answer if all questions arising from it result in a "yes".

Checking for cut-offs is easy: if a new question is asked, we follow the arrows in the reversed direction (to the left). This way, we traverse the list of currently investigated questions. If we meet the question again, we have to cut off the search.

To find the parsings, every question that is answered affirmatively has to pass back a list of rules that start the derivation asked for in the question. This list can be placed into the ellipse, together with the question. We have not done so in Figure 4.3, because it is complicated enough as it is. However, if we strip Figure 4.3 of its dead ends, and leave out the boxes, we get Figure 4.4. In this case, every ellipse only has

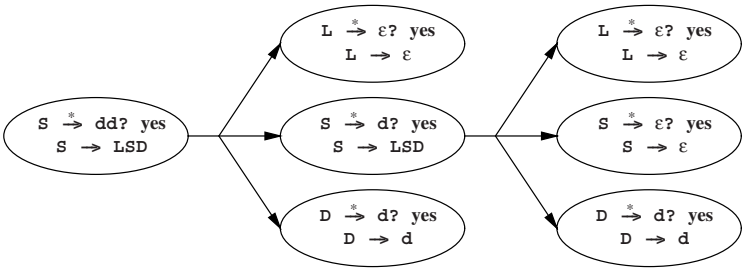


Fig. 4.4. The result of Unger's parser for the sentence `dd`

one possible grammar rule. Therefore, there is only one parsing, and we obtain it by reading Figure 4.4 from left to right, top to bottom:

```
S → LSD → SD → LSDD → SDD → DD → dD → dd
```

In general, the total number of parsings is equal to the product of the number of grammar rules in each ellipse.

This example shows that we can save much time by remembering answers to questions. For example, the question whether `L` derives ε is asked many times. Sheil [20] has shown that the efficiency improves dramatically when this is done: it goes from exponential to polynomial. Another possible optimization is achieved by computing in advance which non-terminals can derive ε. In fact, this is a special case of computing the minimum length of a terminal string that each non-terminal derives. If a non-terminal derives ε, this minimum length is 0.

4.1.3 Getting Parse-Forest Grammars from Unger Parsing

It is surprisingly easy to construct a parse-forest grammar while doing Unger parsing: all that is needed is to add one rule to the parse-forest grammar for each attempted partition. For example, the first partition investigated in Section 4.1.1 (line 6 in Figure 4.2) adds the rule

```
Expr_1_7  →  Expr_1_2 +_3_1 Term_4_4
```

to the parse-forest grammar. Each segment of the partition and the partition itself is designated by a specific non-terminal, the name of which is composed of the original name and the starting point and length of the segment is should produce. This even applies to the original terminals, since the above partition claims that the **+** is specifically the **+** in position 3 (when counting the input tokens starting from 1).

The first partition in Figure 4.2 adds the rule

```
Expr_1_7  →  Expr_1_1 +_2_1 Term_3_5
```

but since the input does not contain a **+_2_1**, a **+** in position 2, the rule can be rejected immediately. Alternatively, one can say that it contains an undefined terminal, and then the grammar clean-up algorithm from Section 2.9.5 will remove it for us. Likewise, the further attempts described in Section 4.1.1 add the rules

```
  Expr_1_2  →  Term_1_2
  Term_1_2  →  Factor_1_2
Factor_1_2  →  i_1_2
```

which again contains an undefined terminal, **i_1_2**. (The first alternative of **Factor**, **Factor→(Expr)**, is not applicable because it requires breaking **Factor_1_2** into three pieces, and we were not yet allowing ε-rules in Section 4.1.1.)

We see that Unger parsing, being a top-down parsing method, creates a lot of undefined non-terminals (and ditto terminals); these represent hypotheses of the top-down process that did not materialize.

```
  Expr_1_7_S  →  Term_1_7
   Term_1_7   →  Term_1_5 ×_6_1 Factor_7_1
   Term_1_5   →  Factor_1_5
Factor_1_5    →  (_1_1 Expr_2_3 )_5_1
   Expr_2_3   →  Expr_2_1 +_3_1 Term_4_1
   Expr_2_1   →  Term_2_1
   Term_2_1   →  Factor_2_1
Factor_2_1    →  i_2_1
   Term_4_1   →  Factor_4_1
Factor_4_1    →  i_4_1
Factor_7_1    →  i_7_1
```

Fig. 4.5. Parse-forest grammar for the string **(i+i)×i**

The parsing process generates a parse-forest grammar of 294 rules, which we do not show here. After clean-up the parse-forest grammar of Figure 4.5 remains, with 11 rules. One sees easily that it is equivalent to the one parsing found for the string **(i+i)×i** at the end of Section 4.1.1.

4.2 The CYK Parsing Method

The parsing method described in this section is attributed to J. Cocke, D.H. Younger, and T. Kasami, who independently discovered variations of the method; it is now known as the *Cocke-Younger-Kasami* method, or the *CYK* method. The most accessible original description is that of Younger [10]. An earlier description is by Sakai [5].

As with Unger's parsing method, the input to the CYK algorithm consists of a CF grammar and an input sentence. The first phase of the algorithm constructs a table telling us which non-terminal(s) derive which substrings of the sentence. This is the recognition phase; it ultimately also tells us whether the input sentence can be derived from the grammar. The second phase uses this recognition table and the grammar to construct all possible derivations of the sentence.

We will first concentrate on the recognition phase, which is the distinctive feature of the algorithm.

4.2.1 CYK Recognition with General CF Grammars

To see how the CYK algorithm solves the recognition and parsing problem, let us consider the grammar of Figure 4.6. This grammar describes the syntax of numbers

```
Number_S  →  Integer | Real
 Integer  →  Digit | Integer Digit
    Real  →  Integer Fraction Scale
Fraction  →  . Integer
   Scale  →  e Sign Integer | Empty
   Digit  →  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    Sign  →  + | -
   Empty  →  ε
```

Fig. 4.6. A grammar describing numbers in scientific notation

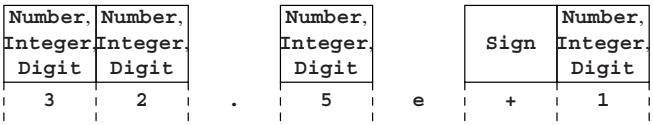
in scientific notation. An example sentence produced by this grammar is `32.5e+1`. We will use this grammar and sentence as an example.

The CYK algorithm first concentrates on substrings of the input sentence, shortest substrings first, and then works its way up. The following derivations of substrings of length 1 can be read directly from the grammar:

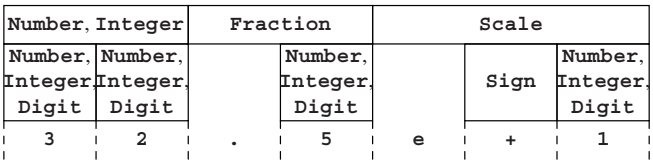
Digit	Digit		Digit		Sign	Digit
3	2	.	5	e	+	1

This means that `Digit` derives `3`, `Digit` derives `2`, etc. Note, however, that this picture is not yet complete. For one thing, there are several other non-terminals deriving `3`. This complication arises because the grammar contains so-called *unit rules*,

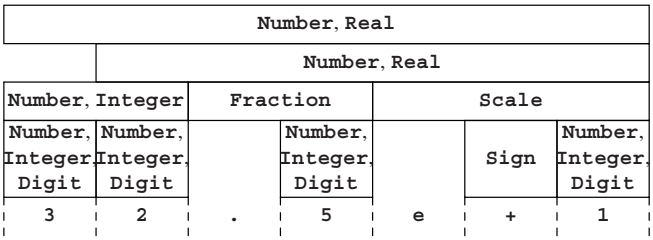
rules of the form $A \rightarrow B$, where A and B are non-terminals. Such rules are also called *single rules* or *chain rules*. We can have chains of them in a derivation. So the next step consists of applying the unit rules, repetitively, for example to find out which other non-terminals derive **3**. This gives us the following result:



Now we already see some combinations that we recognize from the grammar: For example, an **Integer** followed by a **Digit** is again an **Integer**, and a **.** (dot) followed by an **Integer** is a **Fraction**. We get (again also using unit rules):

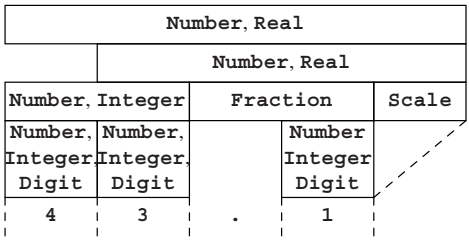


At this point, we see that the rule for **Real** is applicable in several ways, and then the rule for **Number**, so we get:



So we find that **Number** does indeed derive **32.5e+1**.

In the example above, we have seen that unit rules complicate things a bit. Another complication, one that we have avoided until now, is formed by ε-rules. For example, if we want to recognize the input **43.1** according to the example grammar, we have to realize that **Scale** derives ε here, so we get the following picture:



In general this is even more complicated. We must take into account the fact that several non-terminals can derive ε between any two adjacent terminal symbols in the input sentence, and also in front of the input sentence or at the back. However, as we shall see, the problems caused by these kinds of rules can be solved, albeit at a certain cost.

In the meantime, we will not let these problems discourage us. In the example, we have seen that the CYK algorithm works by determining which non-terminals derive which substrings, shortest substrings first. Although we skipped them in the example, the shortest substrings of any input sentence are, of course, the ε-substrings. We shall have to recognize them in arbitrary position, so we first compute R_ε, the set of non-terminals that derive ε, using the following closure algorithm.

The set R_ε is initialized to the set of non-terminals A for which $A \rightarrow \varepsilon$ is a grammar rule. For the example grammar, R_ε is initially the set {**Empty**}. Next, we check each grammar rule: If a right-hand side consists only of symbols that are a member of R_ε, we add the left-hand side to R_ε (it derives ε, because all symbols in the right-hand side do). In the example, **Scale** would be added. This process is repeated until no new non-terminals can be added to the set. For the example, this results in

R_ε = {**Empty**, **Scale**}.

Now we direct our attention to the non-empty substrings of the input sentence. Suppose we have an input sentence $t = t_1 t_2 \cdots t_n$ and we want to compute the set of non-terminals that derive the substring of t starting at position i, of length l. We will use the notation $s_{i,l}$ for this substring, so,

$s_{i,l} = t_i t_{i+1} \cdots t_{i+l-1}$.

or in a different notation: $s_{i,l} = t_{i\ldots i+l-1}$. Figure 4.7 presents this notation graphically, using a sentence of 4 symbols. We will use the notation $R_{i,l}$ for the set of

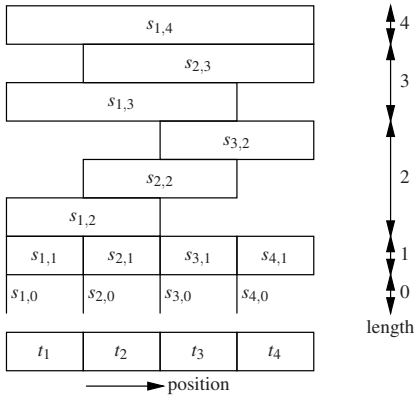
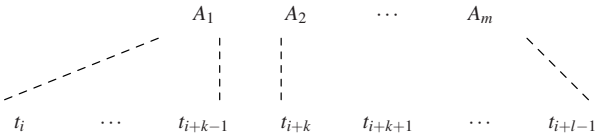


Fig. 4.7. A graphical presentation of substrings

non-terminals deriving the substring $s_{i,l}$. This notation can be extended to deal with substrings of length 0: $s_{i,0} = \varepsilon$, and $R_{i,0} = R_\varepsilon$, for all i.

Because shorter substrings are dealt with first, we can assume that we are at a stage in the algorithm where all information on substrings with length smaller than a certain l is available. Using this information, we check each right-hand side in the grammar, to see if it derives $s_{i,l}$, as follows: suppose we have a right-hand side $A_1 \cdots A_m$. Then we divide $s_{i,l}$ into m (possibly empty) segments, such that A_1 derives the first segment, A_2 the second, etc. We start with A_1. If $A_1 \cdots A_m$ is to derive $s_{i,l}$, A_1 has to derive a first part of it, say of length k. That is, A_1 must derive $s_{i,k}$ (be a member of $R_{i,k}$), and $A_2 \cdots A_m$ must derive the rest:



This is attempted for every k for which A_1 is a member of $R_{i,k}$, including $k = 0$. Naturally, if A_1 is a terminal, then A_1 must be equal to t_i, and k is 1. Checking if $A_2 \cdots A_m$ derives $t_{i+k} \cdots t_{i+l-1}$ is done in the same way. Unlike Unger's method, we do not have to try all partitions, because we already know which non-terminals derive which substrings.

Nevertheless, there are two problems with this. In the first place, m could be 1 and A_1 a non-terminal, so we are dealing with a unit rule. In this case, A_1 must derive the whole substring $s_{i,l}$, and thus be a member of $R_{i,l}$, which is the set that we are computing now, so we do not know yet if this is the case. This problem can be solved by observing that if A_1 is to derive $s_{i,l}$, somewhere along the derivation there must be a first step not using a unit rule. So we have:

$$A_1 \rightarrow B \rightarrow \cdots \rightarrow C \overset{*}{\rightarrow} s_{i,l}$$

where C is the first non-terminal using a non-unit rule in the derivation. Disregarding ε-rules (the second problem) for a moment, this means that at a certain moment in the process of computing the set $R_{i,l}$, C will be added to $R_{i,l}$. Now, if we repeat the computation of $R_{i,l}$ again and again, at some moment B will be added, and during the next repetition, A_1 will be added. So we have to repeat the process until no new non-terminals are added to $R_{i,l}$. This, like the computation of R_ε, is an example of a closure algorithm.

The second problem is caused by the ε-rules. If all but one of the A_t derive ε, we have a problem that is basically equivalent to the problem of unit rules. It too requires recomputation of the entries of R until nothing changes any more, again using a closure algorithm.

In the end, when we have computed all the $R_{i,l}$, the recognition problem is solved: the start symbol S derives $t (= s_{1,n})$ if and only if S is a member of $R_{1,n}$.

This is a complicated process, where part of this complexity stems from the ε-rules and the unit rules. Their presence forces us to do the $R_{i,l}$ computation repeatedly; this is inefficient, because after the first computation of $R_{i,l}$ recomputations yield little new information.

Another less obvious but equally serious problem is that a right-hand side may consist of arbitrarily many non-terminals, and trying all possibilities can be a lot of work. We can see that as follows. For a rule whose right-hand side consists of m members, $m-1$ segment ends have to be found, each of them combining with all the previous ones. Finding a segment end costs $O(n)$ actions, since a list proportional to the length of the input has to be scanned; so finding the required $m-1$ segment ends costs $O(n^{m-1})$. And since there are $O(n^2)$ elements in R, filling it completely costs $O(n^{m+1})$, so the time requirement is exponential in the maximum length of the right-hand sides in the grammar. The longest right-hand side in Figure 4.6 is 3, so the time requirement is $O(n^4)$. This is far more efficient than exhaustive search, which needs a time that is exponential in the length of the input sentence, but still heavy enough to worry about.

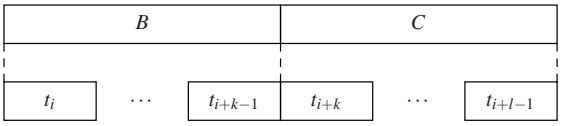
Imposing certain restrictions on the rules may solve these problems to a large extent. However, these restrictions should not limit the generative power of the grammar significantly.

4.2.2 CYK Recognition with a Grammar in Chomsky Normal Form

Two of the restrictions that we want to impose on the grammar are obvious by now: no unit rules and no ε-rules. We would also like to limit the maximum length of a right-hand side to 2; this would reduce the time complexity to $O(n^3)$. It turns out that there is a form for CF grammars that exactly fits these restrictions: the Chomsky Normal Form. It is as if this normal form was invented for this algorithm. A grammar is in *Chomsky Normal Form* (*CNF*), when all rules either have the form $A \rightarrow a$, or $A \rightarrow BC$, where a is a terminal and A, B, and C are non-terminals. Fortunately, as we shall see later, any CF grammar can be mechanically transformed into a CNF grammar.

We will first discuss how the CYK algorithm works for a grammar in CNF. There are no ε-rules in a CNF grammar, so R_ε is empty. The sets $R_{i,1}$ can be read directly from the rules: they are determined by the rules of the form $A \rightarrow a$. A rule $A \rightarrow BC$ can never derive a single terminal, because there are no ε-rules.

Next, we proceed iteratively as before, first processing all substrings of length 2, then all substrings of length 3, etc. When a right-hand side BC is to derive a substring of length l, B has to derive the first part (which is non-empty), and C the rest (also non-empty).



So B must derive $s_{i,k}$, that is, B must be a member of $R_{i,k}$, and likewise C must derive $s_{i+k,l-k}$; that is, C must be a member of $R_{i+k,l-k}$. Determining if such a k exists is easy: just try all possibilities; they range from 1 to $l-1$. All sets $R_{i,k}$ and $R_{i+k,l-k}$ have already been computed at this point.

This process is much less complicated than the one we saw before, which worked with a general CF grammar, for two reasons. The most important one is that we do not have to repeat the process again and again until no new non-terminals are added to $R_{i,l}$. Here, the substrings we are dealing with are really substrings: they cannot be equal to the string we started out with. The second reason is that we have to find only one place where the substring must be split in two, because the right-hand side consists of only two non-terminals. In ambiguous grammars, there can be several different splittings, but at this point that does not worry us. Ambiguity is a parsing issue, not a recognition issue.

The algorithm results in a complete collection of sets $R_{i,l}$. The sentence t consists of only n symbols, so a substring starting at position i can never have more than $n+1-i$ symbols. This means that there are no substrings $s_{i,l}$ with $i+l > n+1$. Therefore, the $R_{i,l}$ sets can be organized in a triangular table, as depicted in Figure 4.8. This table is called the *recognition table*, or the *well-formed substring table*.

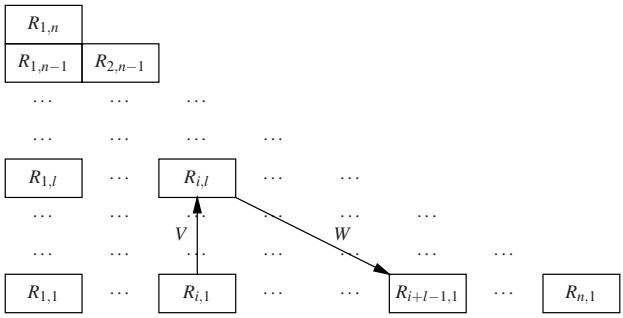
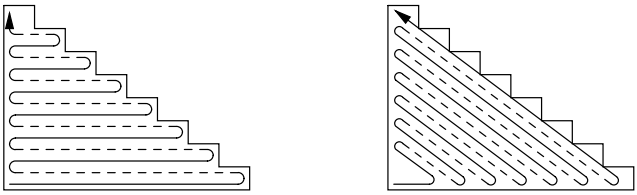


Fig. 4.8. Form of the recognition table

The entry $R_{i,l}$ is computed from entries along the arrows V and W simultaneously, as follows. The first entry we consider is $R_{i,1}$, at the start of arrow V. All non-terminals B in $R_{i,1}$ produce substrings which start at position i and have a length 1. Since we are trying to obtain parsings for the substring starting at position i with length l, we are now interested in substrings starting at $i+1$ and having length $l-1$. These should be looked for in $R_{i+1,l-1}$, at the start of arrow W. Now we combine each of the Bs found in $R_{i,1}$ with each of the Cs found in $R_{i+1,l-1}$, and for each pair B and C for which there is a rule $A \rightarrow BC$ in the grammar, we insert A in $R_{i,l}$. Likewise a B in $R_{i,2}$ can be combined into an A with a C from $R_{i+2,l-2}$, etc., and we continue in this way until we reach $R_{i,l-1}$ at the end point of V and $R_{i+l-1,1}$ at the end of W.

The entry $R_{i,l}$ cannot be computed until all entries below it are known in the triangle of which it is the top. This restricts the order in which the entries can be computed but still leaves some freedom. One way to compute the recognition table is depicted in Figure 4.9(a); it follows our earlier description in which no substring of length l is recognized until all string of length $l-1$ have been recognized. We could also compute the recognition table in the order depicted in Figure 4.9(b). In this order, $R_{i,l}$ is computed as soon as all sets and input symbols needed for its computation are



(*a*) off-line order (*b*) on-line order

Fig. 4.9. Different orders in which the recognition table can be computed

available. This order is particularly suitable for on-line parsing, where the number of symbols in the input is not known in advance, and additional information is computed each time a new symbol is read.

Now let us examine the cost of this algorithm. Figure 4.8 shows that there are $n(n+1)/2$ entries to be filled. Filling in an entry requires examining all entries on the arrow V, of which there are at most n; usually there are fewer, and in practical situations many of the entries are empty and need not be examined at all. We will call the number of entries that really have to be considered n_{occ} for "number of occurrences"; it is usually much smaller than n and for many grammars it is even a constant, but for worst-case estimates it should be replaced by n. Once the entry on the arrow v has been chosen, the corresponding entry on the arrow W is fixed, so the cost of finding it does not depend on n. As a result the algorithm has a time requirement of $O(n^2 n_{occ})$ and operates in a time proportional to the cube of the length of the input sentence in the worst case, as already announced at the beginning of this section.

The cost of the algorithm also depends on the properties of the grammar. The entries along the V and W arrows can each contain at most $|V_N|$ non-terminals, where $|V_N|$ is the number of non-terminals in the grammar, the size of the set V_N from the formal definition of a grammar in Section 2.2. But again the actual number is usually much lower, since usually only a very limited subset of the non-terminals can produce a segment of the input of a given length in a given position; we will indicate the number by $|V_N|_{occ}$. So the cost of one combination step is $O(|V_N|^2_{occ})$. Finding the rule in the grammar that combines B and C into an A can be done in constant time, by hashing or precomputation, and does not add to the cost of one combination step. This gives an overall time requirement of $O(|V_N|^2_{occ} n^2 n_{occ})$.

There is some disagreement in the literature over whether the second index of the recognition table should represent the length or the end position of the recognized segment. It is obvious that both carry the same information, but sometimes one is more convenient and at other times the other. There is some evidence, from Earley parsing (Section 7.2) and parsing as intersection (Chapter 13), that using the end point is more fundamental, but for CYK parsing the length is more convenient, both conceptually and for drawing pictures.

4.2.3 Transforming a CF Grammar into Chomsky Normal Form

The previous section has demonstrated that it is certainly worthwhile to try to transform a general CF grammar into CNF. In this section, we will discuss this transformation, using our number grammar as an example. The transformation is split up into several stages:

- first, ε-rules are eliminated;
- then, unit rules are eliminated;
- then, the grammar is cleaned as described in Section 2.9.5 (optional);
- then, finally, the remaining grammar rules are modified, and rules are added, until they all have the desired form, that is, either $A \rightarrow a$ or $A \rightarrow BC$.

All these transformations will not change the language defined by the grammar. This is not proven here. Most books on formal language theory discuss these transformations more formally and provide proofs; see for example Hopcroft and Ullman [391].

4.2.3.1 Eliminating ε-Rules

Suppose we have a grammar G, with an ε-rule $A \rightarrow \varepsilon$, and we want to eliminate this rule. We cannot just remove the rule, as this would change the language defined by the non-terminal A, and also probably the language defined by the grammar G. So something has to be done about the occurrences of A in the right-hand sides of the grammar rules. Whenever A occurs in a grammar rule $B \rightarrow \alpha A \beta$, we replace this rule with two others: $B \rightarrow \alpha A' \beta$, where A' is a new non-terminal, for which we shall add rules later (these rules will be the non-empty grammar rules of A), and $B \rightarrow \alpha\beta$, which handles the case where A derives ε in a derivation using the $B \rightarrow \alpha A \beta$ rule. Notice that the α and β in the rules above could also contain A; in this case, each of the new rules must be replaced in the same way, and this process must be repeated until all occurrences of A are removed. When we are through, there will be no occurrence of A left in the grammar.

Every ε-rule must be handled in this way. Of course, during this process new ε-rules may originate. This is only to be expected: the process makes all ε-derivations explicit. The newly created ε-rules must be dealt with in exactly the same way. Ultimately this process will stop, because the number of non-terminals that derive ε is finite and in the end none of these non-terminals occur in any right-hand side any more.

The next step in eliminating the ε-rules is the addition of grammar rules for the new non-terminals. If A is a non-terminal for which an A' was introduced, we add a rule $A' \rightarrow \alpha$ for all non-ε-rules $A \rightarrow \alpha$. Since all ε-rules have been made explicit, we can be sure that if a rule does not derive ε directly, it cannot do so indirectly. A problem that may arise here is that there may not be a non-ε-rule for A. In this case, A only derives ε, so we remove all rules using A'.

All this leaves us with a grammar that still contains ε-rules. However, none of the non-terminals having an ε-rule is reachable from the start symbol, with one important

exception: the start symbol itself. In particular, we now have a rule $S \rightarrow \varepsilon$ if and only if ε is a member of the language defined by the grammar G. All other non-terminals with ε-rules can be removed safely, but the actual cleaning up of the grammar is left for later.

The grammar of Figure 4.10 is a nasty grammar to test your ε-rule elimination scheme on. Our scheme transforms this grammar into the grammar of Figure 4.11.

```
S_S  ->  L a M
 L   ->  L M
 L   ->  ε
 M   ->  M M
 M   ->  ε
```

Fig. 4.10. An example grammar to test ε-rule elimination schemes

This grammar still has ε-rules, but these can be eliminated by the removal of non-

```
S_S  ->  L' a M'  |  a M'  |  L' a  |  a
 L   ->  L' M'  |  L'  |  M'  |  ε
 M   ->  M' M'  |  M'  |  ε
L'   ->  L' M'  |  L'  |  M'
M'   ->  M' M'  |  M'
```

Fig. 4.11. Result after our ε-rule elimination scheme

productive and/or unreachable non-terminals. Cleaning up this grammar leaves only one rule: $S \rightarrow a$. Removing the ε-rules in our number grammar results in the grammar of Figure 4.12. Note that the two rules to produce ε, **`Empty`** and **`Scale`**, are still present but are not used any more.

```
 Number_S  ->  Integer | Real
 Integer   ->  Digit | Integer Digit
    Real   ->  Integer Fraction Scale' | Integer Fraction
Fraction   ->  . Integer
  Scale'   ->  e Sign Integer
   Scale   ->  e Sign Integer | ε
   Digit   ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    Sign   ->  + | -
   Empty   ->  ε
```

Fig. 4.12. Our number grammar after elimination of ε-rules

4.2.3.2 Eliminating Unit Rules

The next trouble-makers to be eliminated are the unit rules, that is, rules of the form $A \to B$. It is important to realize that, if such a rule $A \to B$ is used in a derivation, it must be followed at some point by the use of a rule $B \to \alpha$. Therefore, if we have a rule $A \to B$, and the rules for B are

$$B \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n,$$

we can replace the rule $A \to B$ with

$$A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n.$$

In doing this, we can of course introduce new unit rules. In particular, when repeating this process, we could at some point again get the rule $A \to B$. In this case, we have an infinitely ambiguous grammar, because this means that B derives B. Now this may seem to pose a problem, but we can just leave such a unit rule out; the effect is that we short-cut derivations like

$$A \to B \to \cdots \to B \to \cdots$$

Also rules of the form $A \to A$ are left out. In fact, a pleasant side effect of removing ε-rules and unit rules is that the resulting grammar is not infinitely ambiguous any more.

Removing the unit rules in our ε-free number grammar results in the grammar of Figure 4.13.

```
Number_S  ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Number_S  ->  Integer Digit
Number_S  ->  Integer Fraction Scale' | Integer Fraction
Integer   ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Integer   ->  Integer Digit
   Real   ->  Integer Fraction Scale' | Integer Fraction
Fraction  ->  . Integer
  Scale'  ->  e Sign Integer
   Scale  ->  e Sign Integer | ε
   Digit  ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    Sign  ->  + | -
   Empty  ->  ε
```

Fig. 4.13. Our number grammar after eliminating unit rules

4.2.3.3 Cleaning up the Grammar

Although our number grammar does not contain non-productive non-terminals, it does contain unreachable ones, produced by eliminating the ε-rules: **`Real`**, **`Scale`**, and **`Empty`**. The CYK algorithm will work equally well with or without them, so

cleaning up the grammar, as described in Section 2.9.5, is optional. For conceptual and descriptional simplicity we will clean up our grammar here, but further on (Section 4.2.6) we shall see that this is not always advantageous. The cleaned-up grammar is shown in Figure 4.14.

```
  Numbers  ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  Numbers  ->  Integer Digit
  Numbers  ->  Integer Fraction Scale' | Integer Fraction
  Integer  ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  Integer  ->  Integer Digit
Fraction   ->  . Integer
   Scale'  ->  e Sign Integer
   Digit   ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    Sign   ->  + | -
```

Fig. 4.14. Our cleaned-up number grammar

4.2.3.4 Finally, to the Chomsky Normal Form

After all these grammar transformations, we have a grammar without ε-rules or unit rules, all non-terminal are reachable, and there are no non-productive non-terminals. So we are left with two types of rules: rules of the form $A \rightarrow a$, which are already in the proper form, and rules of the form $A \rightarrow X_1 X_2 \cdots X_m$, with $m \geq 2$. For every terminal b occurring in such a rule we create a new non-terminal T_b with only the rule $T_b \rightarrow b$, and we replace each occurrence of b in a rule $A \rightarrow X_1 X_2 \cdots X_m$ with T_b. Now the only rules not yet in CNF are of the form $A \rightarrow X_1 X_2 \cdots X_m$, with $m \geq 3$, and all X_i non-terminals. These rules can now just be split up:

$$A \rightarrow X_1 X_2 \cdots X_m$$

is replaced by the following two rules:

$$\begin{aligned} A &\rightarrow A_1 X_3 \cdots X_m \\ A_1 &\rightarrow X_1 X_2 \end{aligned}$$

where A_1 is a new non-terminal. Now we have replaced the original rule with one that is one shorter, and one that is in CNF. This splitting can be repeated until all parts are in CNF. Figure 4.15 represents our number grammar in CNF.

4.2.4 The Example Revisited

Now let us see how the CYK algorithm works with our example grammar, which we have just transformed into CNF. Again, our input sentence is `32.5e+1`. The recognition table is given in Figure 4.16. The bottom row is read directly from the grammar. For example, the only non-terminals having a production rule with right-hand side `3` are **`Number`**, **`Integer`**, and **`Digit`**. Notice that for each symbol a in

```
Number_S  ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Number_S  ->  Integer Digit
Number_S  ->  N1 Scale' | Integer Fraction
      N1  ->  Integer Fraction
 Integer  ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 Integer  ->  Integer Digit
Fraction  ->  T1 Integer
      T1  ->  .
  Scale'  ->  N2 Integer
      N2  ->  T2 Sign
      T2  ->  e
   Digit  ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    Sign  ->  + | -
```

Fig. 4.15. Our number grammar in CNF

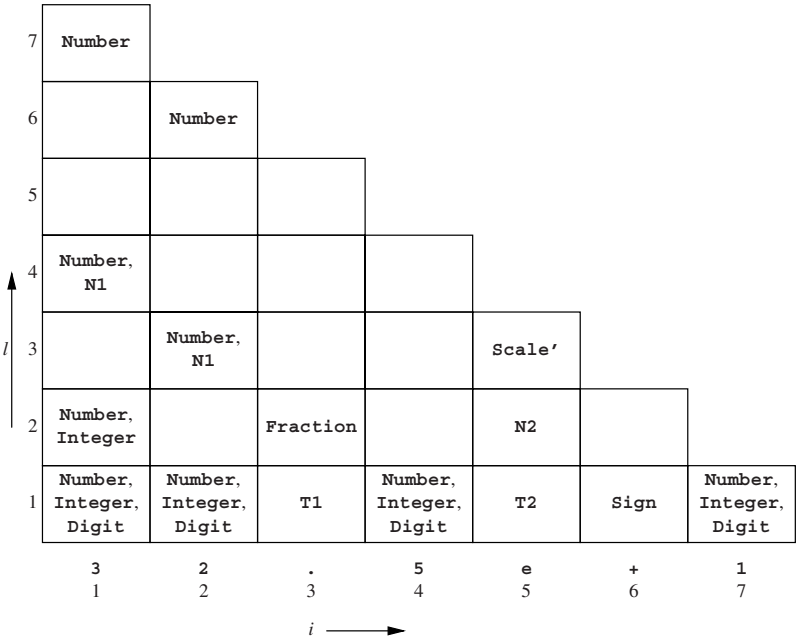


Fig. 4.16. The recognition table for the input sentence `32.5e+1`

the sentence there must be at least one non-terminal A with a production rule $A \rightarrow a$, or else the sentence cannot be derived from the grammar.

The other rows are computed as described before. Actually, there are two ways to compute a certain $R_{i,l}$. The first method is to check each right-hand side in the grammar. For example, to check whether the right-hand side **`N1 Scale'`** derives the substring **`2.5e`** (= $s_{2,4}$). The recognition table derived so far tells us that

- **`N1`** is not a member of $R_{2,1}$ or $R_{2,2}$,

- **N1** is a member of $R_{2,3}$, but **Scale'** is not a member of $R_{5,1}$

so the answer is no. Using this method, we have to check each right-hand side in this way, adding the left-hand side to $R_{2,4}$ if we find that the right-hand side derives $s_{2,4}$.

The second method is to compute possible right-hand sides from the recognition table computed so far. For example, $R_{2,4}$ is the set of non-terminals that have a right-hand side AB where either

- A is a member of $R_{2,1}$ and B is a member of $R_{3,3}$, or
- A is a member of $R_{2,2}$ and B is a member of $R_{4,2}$, or
- A is a member of $R_{2,3}$ and B is a member of $R_{5,1}$.

This gives as possible combinations for AB: **N1 T2** and **Number T2**. Now we check all rules in the grammar to see if they have a right-hand side that is a member of this set. If so, the left-hand side is added to $R_{2,4}$.

4.2.5 CYK Parsing with Chomsky Normal Form

We now have an algorithm that determines whether a sentence belongs to a language or not, and it is much faster than exhaustive search. Most of us, however, not only want to know whether a sentence belongs to a language, but also, if so, how it can be derived from the grammar. If it can be derived in more than one way, we probably want to know all possible derivations. As the recognition table contains the information on all derivations of substrings of the input sentence that we could possibly make, it also contains the information we want. Unfortunately, this table contains too much information, so much that it hides what we want to know. The table may contain information about non-terminals deriving substrings, where these derivations cannot be used in the derivation of the input sentence from the start symbol S. For example, in the example above, $R_{2,3}$ contains **N1**, but the fact that **N1** derives **2.5** cannot be used in the derivation of **32.5e+1** from **Number**.

The key to the solution of this problem lies in the simple observation that the derivation must start with the start symbol S. The first step of the derivation of the input sentence t, with length n, can be read from the grammar, together with the recognition table. If $n = 1$, there must be a rule $S \rightarrow t$; if $n \geq 2$, we have to examine all rules $S \rightarrow AB$, where A derives the first k symbols of t, and B the rest, that is, A is a member of $R_{1,k}$ and B is a member of $R_{k+1,n-k}$, for some k. There must be at least one such rule, or else S would not derive t.

Now, for each of these combinations AB we have the same problem: how does A derive $s_{1,k}$ and B derive $s_{k+1,n-k}$? These problems are solved in exactly the same way. It does not matter which non-terminal is examined first. Consistently taking the leftmost one results in a leftmost derivation, consistently taking the rightmost one results in a rightmost derivation.

Notice that we can use an Unger-style parser for this. However, it would not have to generate all partitions any more, because we already know which partitions will work.

Let us try to find a leftmost derivation for the example sentence and grammar, using the recognition table of Figure 4.16. We begin with the start symbol, **Number**.

Our sentence contains seven symbols, which is certainly more than one, so we have to use one of the rules with a right-hand side of the form *AB*. The **Integer Digit** rule is not applicable here, because the only instance of **Digit** that could lead to a derivation of the sentence is the one in $R_{7,1}$, but **Integer** is not a member of $R_{1,6}$. The **Integer Fraction** rule is not applicable either, because there is no **Fraction** deriving the last part of the sentence. This leaves us with the production rule **Number → N1 Scale'**, which is indeed applicable, because **N1** is a member of $R_{1,4}$, and **Scale'** is a member of $R_{5,3}$, so **N1** derives **32.5** and **Scale'** derives **e+1**.

Next, we have to find out how **N1** derives **32.5**. There is only one applicable rule: **N1 → Integer Fraction**, and it is indeed applicable, because **Integer** is a member of $R_{1,2}$, and **Fraction** is a member of $R_{3,2}$, so **Integer** derives **32**, and **Fraction** derives **.5**. In the end, we find the following derivation:

```
Number →
N1 Scale' →
Integer Fraction Scale' →
Integer Digit Fraction Scale' →
3 Digit Fraction Scale' →
3 2 Fraction Scale' →
3 2 T1 Integer Scale' →
3 2 . Integer Scale' →
3 2 . 5 Scale' →
3 2 . 5 N2 Integer →
3 2 . 5 T2 Sign Integer →
3 2 . 5 e Sign Integer →
3 2 . 5 e + Integer →
3 2 . 5 e + 1
```

Unfortunately, this is not exactly what we want, because this is a derivation that uses the rules of the grammar of Figure 4.15, not the rules of the grammar of Figure 4.6, the one that we started with.

4.2.6 Undoing the Effect of the CNF Transformation

When we examine the grammar of Figure 4.6 and the recognition table of Figure 4.16, we see that the recognition table contains the information we need on most of the non-terminals of the original grammar. However, there are a few non-terminals missing in the recognition table: **Scale**, **Real**, and **Empty**. **Scale** and **Empty** were removed because they became unreachable, after the elimination of ε-rules. **Empty** was removed altogether, because it only derived the empty string, and **Scale** was replaced by **Scale'**, where **Scale'** derives exactly the same as **Scale**, except for the empty string. We can use this to add some more information to the recognition table: at every occurrence of **Scale'**, we add **Scale**.

The non-terminal **Real** was removed because it became unreachable after eliminating the unit rules. Now, the CYK algorithm does not *require* that all non-terminals in the grammar be reachable. We could just as well have left the non-terminal **Real**

in the grammar, and transformed its rules to CNF. The CYK algorithm would then have added **Real** to the recognition table, wherever that would be appropriate. The rules for **Real** that would be added to the grammar of Figure 4.15 are:

```
Real  ->  N1 Scale'  |  Integer Fraction
```

The resulting recognition table is presented in Figure 4.17. In this figure, we have

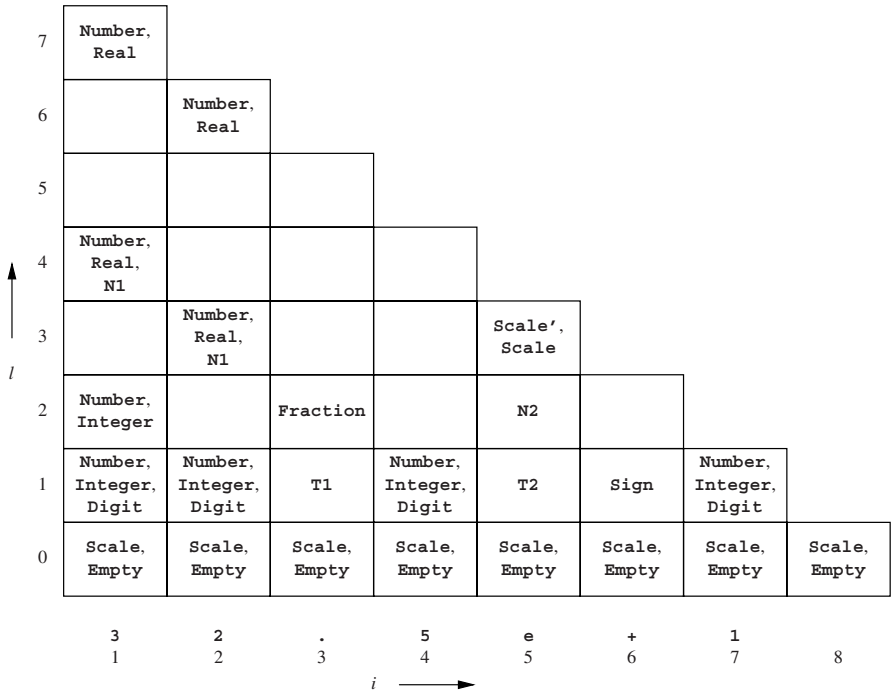


Fig. 4.17. The recognition table with **Scale**, **Real**, and **Empty** added

also added an extra row at the bottom of the triangle. This extra row represents the non-terminals that derive the empty string. These non-terminals can be considered as possibly occurring between any two adjacent symbols in the sentence, and also in front of or at the end of the sentence. The set $R_{i,0}$ represents the non-terminals that can be considered as possibly occurring in front of symbol t_i and the set $R_{n+1,0}$ represents the ones that can occur at the end of the sentence.

Now we have a recognition table which contains all the information we need to parse a sentence with the original grammar. Again, a derivation starts with the start symbol S. If $A_1A_2\cdots A_m$ is a right-hand side of S, we want to know if this rule can be applied, that is, if $A_1A_2\cdots A_m$ derives $s_{1,n}$. This is checked, starting with A_1. There are two cases:

- A_1 is a terminal symbol. In this case, it must be the first symbol of $s_{1,n}$, or this rule is not applicable. Then, we must check if $A_2 \cdots A_m$ derives $s_{2,n-1}$, in the same way that we are now checking if $A_1 A_2 \cdots A_m$ derives $s_{1,n}$.
- A_1 is a non-terminal. In this case, it must be a member of a $R_{1,k}$, for some k, or this rule is not applicable. Then, we must check if $A_2 \cdots A_m$ derives $s_{k+1,n-k}$, in the same way that we are now checking if $A_1 A_2 \cdots A_m$ derives $s_{1,n}$. If we want all parsings, we must do this for every k for which A_1 is a member of $R_{1,k}$. Notice that non-terminals deriving the empty string pose no problem at all, because they appear as a member of $R_{i,0}$ for all i.

We have now determined whether the rule is applicable, and if it is, which parts of the rule derive which substrings. The next step now is to determine how the substrings can be derived. These tasks are similar to the task we started with, and are solved in the same way. This process will terminate at some time, provided the grammar does not contain loops. This is simply an Unger parser which knows in advance which partitions will lead to a successful parse.

Let us go back to the grammar of Figure 4.6 and the recognition table of Figure 4.17, and see how this works for our example input sentence. We now know that **`Number`** derives **`32.5e+1`**, and want to know how. We first ask ourselves: can we use the **`Number→Integer`** rule? **`Integer`** is a member of $R_{1,1}$ and $R_{1,2}$, but there is nothing behind the **`Integer`** in the rule to derive the rest of the sentence, so we cannot use this rule. Can we use the **`Number→Real`** rule? Yes we can, because **`Real`** is a member of $R_{1,7}$, and the length of the sentence is 7. So we start our derivation with

`Number → Real →` $\cdots$

Now we get similar questions for the **`Real`** non-terminal: can we use the **`Real → Integer Fraction Scale`** rule? Well, **`Integer`** is a member of $R_{1,1}$, but we cannot find a **`Fraction`** in any of the $R_{2,k}$ sets. However, **`Integer`** is also a member of $R_{1,2}$, and **`Fraction`** is a member of $R_{3,2}$. Now, **`Scale`** is a member of $R_{5,0}$; this does not help because it would leave nothing in the rule to derive the rest. Fortunately, **`Scale`** is also a member of $R_{5,3}$, and that matches exactly to the end of the string. So this rule is indeed applicable, and we continue our derivation:

`Number → Real → Integer Fraction Scale →` $\cdots$

The sentence is now split up into three parts:

`Number`		
`Real`		
`Integer`	**`Fraction`**	**`Scale`**
`3 2`	`. 5`	`e + 1`

It is left to the reader to verify that we will find only one derivation, and that this is it:

```
Number ->
Real ->
Integer Fraction Scale ->
Integer Digit Fraction Scale ->
Digit Digit Fraction Scale ->
3 Digit Fraction Scale ->
3 2 Fraction Scale ->
3 2 . Integer Scale ->
3 2 . Digit Scale ->
3 2 . 5 Scale ->
3 2 . 5 e Sign Integer ->
3 2 . 5 e + Integer ->
3 2 . 5 e + Digit ->
3 2 . 5 e + 1
```

4.2.7 A Short Retrospective of CYK

We have come a long way. We started with building a recognition table using the original grammar. Then we found that using the original grammar with its unit rules and ε-rules is somewhat complicated, although it can certainly be done. We proceeded by transforming the grammar to CNF. CNF does not contain unit rules or ε-rules. Our gain in this respect was that the algorithm for constructing the recognition table became much simpler. The limitation of the maximum length of a right-hand side to 2 was a gain in efficiency, and also a little in simplicity. However, Sheil [20] has demonstrated that the efficiency only depends on the maximum number of non-terminals occurring in a right-hand side of the grammar, not on the length of the right-hand sides per sé. This can easily be understood, once one realizes that the efficiency depends on (among other things) the number of cuts in a substring that are "difficult" to find, when checking whether a right-hand side derives this substring. This number of "difficult" cuts only depends on the number of non-terminals in the right-hand side. So, for efficiency, Chomsky Normal Form is a bit too restrictive.

A disadvantage of this transformation to CNF is that the resulting recognition table lacks some information that we need to construct a derivation using the original grammar. In the transformation process, some non-terminals were thrown away, because they became non-productive. Fortunately, the missing information could easily be recovered. Ultimately, this process resulted in almost the same recognition table that we would get with our first attempt using the original grammar. It only contains some extra information on non-terminals that were added during the transformation of the grammar to CNF. More importantly, however, it was obtained in a simpler and much more efficient way.

For a more elaborate version of the CYK algorithm, applied to Tree Adjoining Grammars, see Section 15.4.2.

4.2.8 Getting Parse-Forest Grammars from CYK Parsing

As with Unger parsers, it is quite simple to obtain a parse-forest grammar during CYK parsing: whenever a non-terminal A is entered into entry $R_{i,l}$ of the recognition table because there is a rule $A \rightarrow BC$ and B is in $R_{i,k}$ and C is in $R_{i+k,l-k}$, the rule

$$A\_i\_l \quad \rightarrow \quad B\_i\_k \;\; C\_m\_n$$

is added to the parse-forest grammar, where $m = i+k$ and $n = i+l-k$.

There are two things to note here. The first is that this algorithm will never introduce undefined non-terminals: a rule $A\_i\_l \rightarrow B\_i\_k \;\; C\_m\_n$ is added only when it is guaranteed that rules for B_i_k and C_m_n exist. On the other hand, such a rule is added without any regard to its reachability: the non-terminal A_i_l may be reachable from the start symbol or it may not; it is just too early to tell. We see that CYK parsing, being a bottom-up algorithm, creates a lot of unreachable non-terminals; these represent finds of the bottom-up process that led nowhere.

The second is that the parse-forest grammar contains more information than the recognition table, since it not only records for each non-terminal in a given entry *that* it is there but also *why* it is there. The parse-forest grammar combines the recognition phase (Section 4.2.2) and the parsing phase (Section 4.2.5) of the CYK parsing process.

Obtaining the parse-forest grammar from Figure 4.17 and the grammar from Figure 4.6 is straightforward; the result is in Figure 4.18. We see that it contains many unreachable non-terminals, for example **`Number_2_6`**, **`Scale_5_0`**, etc. Removing these yields the parse-forest grammar of Figure 4.19; it is easy to see that it is equivalent to the one derivation found at the end of Section 4.2.6.

4.3 Tabular Parsing

We have drawn the CYK recognition tables as two-dimensional triangular matrices, but the complexity of the entries — sets of non-terminals — already shows that this representation is not in its most elementary form. Simplicity and insight can be gained by realizing that a CYK recognition table is a superposition of a number of tables, one for each non-terminal in the grammar; the entries in *these* tables are just bits, saying "Present" or "Not Present". Since the grammar for numbers from Figure 4.6 has 8 non-terminals, the recognition table from Figure 4.17 is a superposition of 8 matrices. They are shown in Figure 4.20. The dot in the top left corner of the table for **`Number`** means that a **`Number`** of length 7 has been recognized in position 1; the one almost at the bottom right corner means that a **`Number`** of length 1 has been recognized in position 7; etc.

Imagine these 8 tables standing upright in the order **`Number`** $\cdots$ **`Empty`**, perhaps cut out of transparent plastic, glued together into a single block. Now topple the block backwards, away from you. A new matrix appears, T, on what was the bottom of the block before you toppled it, as shown in Figure 4.21, where the old recognition table is still visible on what is now the top. The new table still has the

```
      Start_S  ->  Number_1_7
   Number_1_7  ->  Real_1_7
     Real_1_7  ->  Integer_1_2 Fraction_3_2 Scale_5_3
   Number_2_6  ->  Real_2_6
     Real_2_6  ->  Integer_2_1 Fraction_3_2 Scale_5_3
   Number_1_4  ->  Real_1_4
     Real_1_4  ->  Integer_1_2 Fraction_3_2 Scale_5_0
   Number_2_3  ->  Real_2_3
     Real_2_3  ->  Integer_2_1 Fraction_3_2 Scale_5_0
    Scale_5_3  ->  e_5_1 Sign_6_1 Integer_7_1
   Number_1_2  ->  Integer_1_1 Digit_2_1
  Integer_1_2  ->  Integer_1_1 Digit_2_1
 Fraction_3_2  ->  ._3_1 Integer_4_1
   Number_1_1  ->  Integer_1_1
  Integer_1_1  ->  Digit_1_1
    Digit_1_1  ->  3_1_1
   Number_2_1  ->  Integer_2_1
  Integer_2_1  ->  Digit_2_1
    Digit_2_1  ->  2_2_1
   Number_4_1  ->  Integer_4_1
  Integer_4_1  ->  Digit_4_1
    Digit_4_1  ->  5_4_1
     Sign_6_1  ->  +_6_1
   Number_7_1  ->  Integer_7_1
  Integer_7_1  ->  Digit_7_1
    Digit_7_1  ->  1_7_1
    Scale_5_0  ->  Empty_5_0
    Empty_5_0  ->  ε
```

Fig. 4.18. Parse-forest grammar retrieved from Figure 4.17 and the grammar from Figure 4.6

```
      Start_S  ->  Number_1_7
   Number_1_7  ->  Real_1_7
     Real_1_7  ->  Integer_1_2 Fraction_3_2 Scale_5_3
    Scale_5_3  ->  e_5_1 Sign_6_1 Integer_7_1
  Integer_1_2  ->  Integer_1_1 Digit_2_1
 Fraction_3_2  ->  ._3_1 Integer_4_1
  Integer_1_1  ->  Digit_1_1
    Digit_1_1  ->  3_1_1
    Digit_2_1  ->  2_2_1
  Integer_4_1  ->  Digit_4_1
    Digit_4_1  ->  5_4_1
     Sign_6_1  ->  +_6_1
  Integer_7_1  ->  Digit_7_1
    Digit_7_1  ->  1_7_1
```

Fig. 4.19. Cleaned parse-forest grammar obtained by CYK parsing

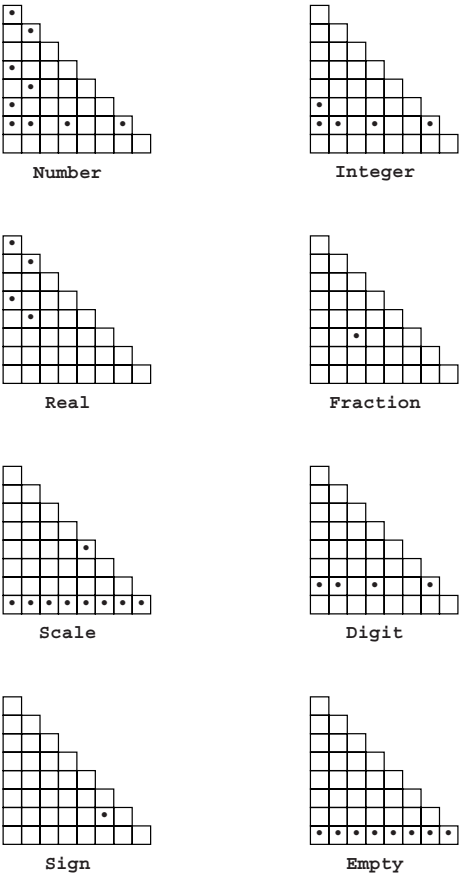


Fig. 4.20. The 8 faces of the table in Figure 4.17

positions in the input as the horizontal axis, but the vertical axis now consists of names of non-terminals, and the entries are lists of lengths. For example, the list {1,2,4,7} in $T_{1,\texttt{Number}}$ in the top left corner means that productions of **`Number`** of these lengths can be recognized in position 1. Parsing algorithms that use mainly this representation are called *tabular parsing algorithms*. It will be clear that no information is gained or lost in this transformation, but the tabular representation has its own advantages and disadvantages.

The table T is initialized by putting a 1 in all entries $T_{i,A}$ where the input has a token t in position i and the grammar has a rule $A \rightarrow t$. There are two ways to fill the rest of the table, top-down and bottom-up.

4.3.1 Top-Down Tabular Parsing

For recognition we are only interested in one element in one entry of the table T: does $T_{\texttt{S},1}$ contain n, where **`S`** is the start symbol and n is the length of the input? To

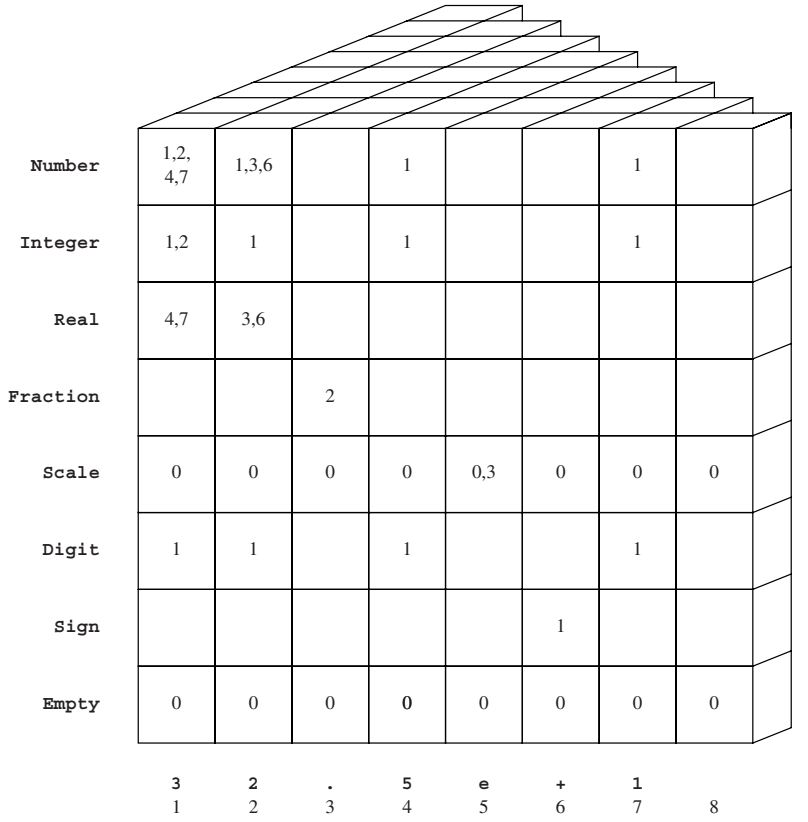


Fig. 4.21. A tabular representation of the table in Figure 4.17

find this out, we stack this query in some form on a stack and, using all grammar rules for **S**, we draw up a list of possibilities for $T_{\mathrm{S},1}$ to contain n, much as we did in the Unger parser. For a rule like **S→AB** these include:

does $T_{\mathrm{A},1}$ contain 0 and does $T_{\mathrm{B},1}$ contain n?
does $T_{\mathrm{A},1}$ contain 1 and does $T_{\mathrm{B},2}$ contain $n-1$?
does $T_{\mathrm{A},1}$ contain 2 and does $T_{\mathrm{B},3}$ contain $n-2$?
...
does $T_{\mathrm{A},1}$ contain n and does $T_{\mathrm{B},n+1}$ contain 0?

If the conditions in any of these lines are fulfilled, $T_{\mathrm{S},1}$ must be made to contain n.

Each of these new queries can be expanded and examined in this same way. In the end the queries will develop into "terminal queries", queries that can be resolved without creating new queries. Examples are "does $T_{\mathtt{a},k}$ contain 1?", which can be answered by checking if the input contains an **a** in position k, and "does $T_{P,k}$ contain 0", which is equivalent to "does P produce ε?". Once we have obtained an answer to a query we store it in the proper position in the table, and we do this not only for

the top query, but also for all intermediate, generated queries. This is very important since now we can obtain the answer without further computation if the same query turns up again, which it frequently does. Note that this requires the possibility to store negative information (actually positive information about absence) in the entries of table T: an entry like $T_{A,k}$ can contain information like "does not contain 7". Also note that this process does not always compute all entries in the table; it may miss entries that, given the grammar, can never be part of a recognition. For some applications this is an advantage.

The technique of storing results of computations in a table in order to replace recomputation by table lookup is called *memoization* (this is not a spelling error; there is really no 'r' in the word). It is a very useful and widely applicable device in algorithmics, and can often reduce the time requirements of an algorithm from exponential to polynomial, as it does in the present example. Memoization was invented in 1968 by Michie [410] and introduced in parsing by Sheil [20], who did not yet use the term "memoization"; see also Norvig [343].

Furthermore we have to concern ourselves with left-recursive non-terminals, again using the same technique as in the Unger parser. If the non-terminal A is left-recursive, the query "does $T_{A,1}$ contain n" will eventually again create the query "does $T_{A,1}$ contain n", and will thus start an endless loop. The loop can easily be prevented by just discarding this recursive occurrence of the same query, since a second computation would not bring in any information not already obtained by the first. Whether a generated query is a recursive occurrence can be determined by looking it up in the stack of queries. In short, top-down tabular parsing is very similar to Unger parsing with memoization.

A full implementation of this algorithm is discussed in Section 17.3.

4.3.2 Bottom-Up Tabular Parsing

The bottom-up tabular parsing algorithm fills all entries correctly, but requires more care than the top-down algorithm. Like CYK, it works most efficiently for grammars that are in Chomsky Normal Form, and we will assume our grammar to be so. And, like in CYK, we need to be careful with the order in which we compute the entries. Also, we will fill the three-dimensional "wedge" of Figure 4.21, the entries of which are Booleans (bits), rather than its two-dimensional front panel, the entries of which are lists of integers. The entry in the wedge which describes whether a terminal production of A with length k starts in position i is written $T_{i,A,k}$.

Bottom-up tabular parsing fills the whole recognition table by filling columns starting from the right end. To fill entry $T_{i,A,k}$, we find a rule of the form $A \rightarrow BC$ from the grammar, and we access $T_{i,B,1}$. If this entry is set, there is a segment at i of length 1 produced by B. So we access $T_{i+1,C,k-1}$, and if it is also set, there is a segment at $i+1$ of length $k-1$ produced by C. From this we conclude that the input segment at i of length k holds a terminal production of A and we set the entry $T_{i,A,k}$. If not, we try again with $T_{i,B,2}$ and $T_{i+2,C,k-2}$, etc., until $T_{i,B,k-1}$ and $T_{i+k-1,C,1}$, just as in the CYK algorithm.

We can stop at $T_{i+k-1,C,1}$, because grammars in CNF have no ε-rules, so $T_{i+k,C,0}$ does not exist. This means that the computation of the entry $T_{i,A,k}$ involves entries $T_{i,B,j}$ with $j < k$ only. So if we compute all entries $T_{i,P,j}$ before all entries $T_{i,Q,k}$ for $j < k$ and arbitrary P and Q, the values of all entries needed for the computation of $T_{i,A,k}$ are guaranteed to be ready. This imposes a particular computation order on the algorithm:

```
for all input positions i from n to 1
    for all non-terminals A in the grammar
        for all length k from 1 to i
            compute T_{i,A,k}
```

The cost of computing one entry $T_{i,A,k}$ is $O(n|P_{av}|)$, where n is the length of the input and $|P_{av}|$ the average number of production rules of a non-terminal. As we saw, this computation is repeated $O(n|G|n)$ times, where $|G|$ is proportional to the size of the grammar. So the time requirements of this parsing algorithm is $O(n^3|G||P_{av}|)$ or $O(n^3) \times O(|G||P_{av}|)$.

Bottom-up tabular parsing is applied in a number of algorithms in Sections 12 and 15.7.

Nederhof and Satta [40] have written a tutorial on tabular parsing, applying it to a wide selection of non-deterministic parsing algorithms.

4.4 Conclusion

The non-directional methods take hold of the input string over its full width. The top-down method (Unger) tries to cut the input string into segments and impose a structure on it deriving from the start symbol; if it succeeds, it has found a parsing. The bottom-up method tries to divide the input into recognizable segments, which can then be assembled into the start symbol; if it succeeds, it has found a parsing. Although their look-and-feel is quite different, both methods can be implemented by filling in a table; only the order in which the entries are computed differs.

Rus [28] presents a remarkable, deterministic non-directional bottom-up parsing algorithm.

Problems

Problem 4.1: A string U is a *supersequence* of a string S if U can be created from S by inserting zero or more tokens from the language in arbitrary places in S. (See also Section 12.4.) a) Design an Unger parser for a grammar G that will recognize a supersequence of a string in the language generated by G. b) Do the same for a CYK parser.

Problem 4.2: A string U is a *subsequence* of a string S if U can be created from S by deleting zero or more tokens from arbitrary places in S. (See also Section 12.4.)

a) Design an Unger parser for a grammar G that will recognize a subsequence of a string in the language generated by G. b) Do the same for a CYK parser.

Problem 4.3: *Project:* Eliminating ε-rules from a grammar greatly modifies it, and effort must be spent to undo the damage during parsing. Much of the effort can be saved by incorporating the removed εs in the modified grammar, as follows. Given a grammar like $\mathtt{S} \rightarrow \mathtt{aBc}$, $\mathtt{B} \rightarrow \mathtt{b} \mid \varepsilon$, we first convert it to AND-OR form, recording the original right-hand side in the name of the non-terminal. (A grammar is in *AND/OR form* if there are only two kinds of rules, AND-rules which specify concatenation of grammar symbols, and OR-rules which specify choice between non-terminal symbols, and there is only one rule for each non-terminal. This names the alternatives rather than the non-terminals.) This yields $\mathtt{S} \rightarrow \mathtt{aBc}$, $\mathtt{B} \rightarrow \mathtt{B_b} \mid \mathtt{B}_\varepsilon$, $\mathtt{B_b} \rightarrow \mathtt{b}$, $\mathtt{B}_\varepsilon \rightarrow \varepsilon$. Next substitute out all nullable OR-rules (the one for B in our case): $\mathtt{S} \rightarrow \mathtt{S}_{\mathtt{aB_bc}} \mid \mathtt{S}_{\mathtt{aB}_\varepsilon\mathtt{c}}$, $\mathtt{S}_{\mathtt{aB_bc}} \rightarrow \mathtt{aB_bc}$, $\mathtt{S}_{\mathtt{aB}_\varepsilon\mathtt{c}} \rightarrow \mathtt{aB}_\varepsilon\mathtt{c}$, $\mathtt{B_b} \rightarrow \mathtt{b}$, $\mathtt{B}_\varepsilon \rightarrow \varepsilon$. Now substitute the rules of the form $A \rightarrow \varepsilon$: $\mathtt{S} \rightarrow \mathtt{S}_{\mathtt{aB_bc}} \mid \mathtt{S}_{\mathtt{aB}_\varepsilon\mathtt{c}}$, $\mathtt{S}_{\mathtt{aB_bc}} \rightarrow \mathtt{aB_bc}$, $\mathtt{S}_{\mathtt{aB}_\varepsilon\mathtt{c}} \rightarrow \mathtt{ac}$, $\mathtt{B_b} \rightarrow \mathtt{b}$. Then when parsing $\mathtt{S}_{\mathtt{aB}_\varepsilon\mathtt{c}} \rightarrow \mathtt{ac}$ the subscript of the S tells us the real form of the right-hand side.
Elaborate this idea into a complete algorithm, with parser.

Problem 4.4: Remove the unit rules from the grammar

$$\begin{array}{lcl} \mathtt{S_S} & \rightarrow & \mathtt{T} \\ \mathtt{T} & \rightarrow & \mathtt{U} \\ \mathtt{U} & \rightarrow & \mathtt{T} \end{array}$$

Problem 4.5: *Research Project:* CYK, especially in its chart parsing form, has long been a favorite of natural language parsing, but we have seen its time requirements are $O(|G||P_{av}|n^3)$. With some natural language grammars being very large (millions of rules), especially the generated ones, even $O(|G|)$ is a problem, regardless of the $O(|P_{av}|)$. Design a version of the CYK/chart algorithm that is better than $O(|G|)$. Do not count on $|\text{ADJ}|$ to be substantially smaller than $|G|^2$, where ADJ is the set of pairs of non-terminals that occur adjacently in any right-hand side. (See also Problem 3.11.)

Problem 4.6: *Project:* When looking at program source code in a programming language, usually seeing two adjacent tokens is enough to get a pretty good idea of which syntactic structure we are looking at. This would eliminate many of the bottom-up hypotheses that CYK maintains. Use this idea to automatically suppress the bulk of the table entries, hopefully leaving only a limited number. This would make it an almost linear-time parser, which might be important for parsing legacy code, which comes with notoriously sloppy grammar.
Try to suppress more hypotheses of the form $A \rightarrow \alpha$ by checking tokens that can occur just before, at the beginning of, inside, at the end of, and just after, a terminal production of A.

Problem 4.7: Formulate the inference rule for the computation of $T_{i,A}$ in Section 4.3.

Problem 4.8: Draw Figures 4.8 and 4.16 using the end position of the recognized segment as the second index.

Problem 4.9: *Project* Determine the class of grammars for which Rus's algorithm [28] works.

Problem 4.10: *Formal Languages*: Design an algorithm to transform a given grammar into one with the lowest possible number of non-terminals. This is important since the time requirements of many parsing algorithms depend on the grammar size.

5

Regular Grammars and Finite-State Automata

Regular grammars, Type 3 grammars, are the simplest form of grammars that still have generative power. They can describe concatenation (joining two strings together) and repetition, and can specify alternatives, but they cannot express nesting. Regular grammars are probably the best-understood part of formal linguistics and almost all questions about them can be answered.

5.1 Applications of Regular Grammars

In spite of their simplicity there are many applications of regular grammars, of which we will briefly mention the most important ones.

5.1.1 Regular Languages in CF Parsing

In some parsers for CF grammars, a subparser can be discerned which handles a regular grammar. Such a subparser is based implicitly or explicitly on the following surprising phenomenon. Consider the sentential forms in leftmost or rightmost derivations. Such sentential forms consist of a closed (finished) part, which contains terminal symbols only and an open (unfinished) part which contains non-terminals as well. In leftmost derivations the open part starts at the leftmost non-terminal and extends to the right; in rightmost derivations the open part starts at the rightmost non-terminal and extends to the left. See Figure 5.1 which uses sample sentential forms from Section 2.4.3.

Fig. 5.1. Open parts in leftmost and rightmost productions

It can easily be shown that these open parts of the sentential form, which play an important role in some CF parsing methods, can be described by a regular grammar, and that that grammar follows from the CF grammar.

To explain this clearly we first have to solve a notational problem. It is conventional to use upper case letters for non-terminals and lower case for terminals, but here we will be writing grammars that produce parts of sentential forms, and since these sentential forms can contain non-terminals, our grammars will have to produce non-terminals. To distinguish these "dead" non-terminals from the "live" ones which do the production, we shall print them barred: $\bar{\mathtt{X}}$.

With that out of the way we can construct a regular grammar *G* with start symbol R for the open parts in leftmost productions of the grammar *C* used in Section 2.4.3, which we repeat here:

```
S_S  →  L & N
 S   →  N
 L   →  N , L
 L   →  N
 N   →  t | d | h
```

The first possibility for the start symbol R of *G* is to produce the start symbol of *C*; so we have R→$\bar{\mathtt{S}}$, where $\bar{\mathtt{S}}$ is just a token. The next step is that this token, being the leftmost non-terminal in the sentential form, is turned into a "live" non-terminal, from which we are going to produce more of the sentential form: R→S. Here S is a non-terminal in *G*, and describes open parts of sentential forms deriving from S in *C*. The first possibility for S in *G* is to produce the right-hand side of S in *C* as tokens: S→$\bar{\mathtt{L}}$&$\bar{\mathtt{N}}$. But it is also possible that $\bar{\mathtt{L}}$, being the leftmost non-terminal in the sentential form, is already alive: S→L&$\bar{\mathtt{N}}$, and it may even have finished producing, so that all its tokens have already become part of the closed part of the sentential form; this leaves &$\bar{\mathtt{N}}$ for the open part: S→&$\bar{\mathtt{N}}$. Next we can move the & from the open part to the closed part: S→$\bar{\mathtt{N}}$. Again this $\bar{\mathtt{N}}$ can become productive: S→N, and, like the L above, can eventually disappear entirely: S→ε. We see how the original S→$\bar{\mathtt{L}}$&$\bar{\mathtt{N}}$ gets gradually worked down to S→ε. The second alternative of S in *C*, S→N, yields the rules S→$\bar{\mathtt{N}}$, S→N, and S→ε, but we had obtained these already.

The above procedure introduces the non-terminals L and N of *G*. Rules for them can be derived in the same way as for S; and so on. The result is the left-regular grammar *G*, shown in Figure 5.2. We have already seen that the process can create

```
R → S̄          L → N̄ , L̄
R → S          L → N , L̄
S → L̄ & N̄      L → , L̄
S → L & N̄      L → L̄
S → & N̄        L → L ✘
S → N̄          L → ε
S → N          L → N̄
S → ε          L → N
N → t | d | h
N → ε
```

Fig. 5.2. A (left-)regular grammar for the open parts in leftmost derivations

duplicate copies of the same rule; we now see that it can also produce loops, for example the rule **L→L**, marked ✘ in the figure. Since such rules contribute nothing, they can be ignored.

In a similar way a right-regular grammar can be constructed for open parts of sentential forms in a rightmost derivation. These grammars are useful for a better understanding of top-down and bottom-up parsing (Chapters 6 and 7) and are essential to the functioning of some parsers (Sections 9.13.2 and 10.2.3).

5.1.2 Systems with Finite Memory

CF (or stronger) grammars allow nesting. Since nesting can, in principle, be arbitrarily deep, the generation of correct CF (or stronger) sentences can require an arbitrary amount of memory to temporarily hold the unprocessed nesting information. Mechanical systems do not possess an arbitrary amount of memory and consequently cannot exhibit CF behavior and are restricted to regular behavior. This is immediately clear for simple mechanical systems like vending machines, traffic lights and DVD recorders: they all behave according to a regular grammar. It is also in principle true for more complicated mechanical systems, like a country's train system or a computer. However, here the argument gets rather vacuous since nesting information can be represented very efficiently and a little memory can take care of a lot of nesting. Consequently, although these systems in principle exhibit regular behavior, it is often easier to describe them with CF or stronger means, even though that incorrectly ascribes infinite memory to them.

Conversely, the global behavior of many systems that do have a lot of memory can still be described by a regular grammar, and many CF grammars are already for a large part regular. This is because regular grammars already take adequate care of concatenation, repetition and choice; context-freeness is only required for nesting. If we call a rule that produces a regular (sub)language (and which consequently could be replaced by a regular rule) "quasi-regular", we can observe the following. If all alternatives of a rule contain terminals only, that rule is quasi-regular (choice). If all alternatives of a rule contain only terminals and non-terminals with quasi-regular and non-recursive rules, then that rule is quasi-regular (concatenation). And if a rule is recursive but recursion occurs only at the end of an alternative and involves only quasi-regular rules, then that rule is again quasi-regular (repetition). This often covers large parts of a CF grammar. See Krzemień and Łukasiewicz [142] for an algorithm to identify all quasi-regular rules in a grammar.

Natural languages are a case in point. Although CF or stronger grammars seem necessary to delineate the set of correct sentences (and they may very well be, to catch many subtleties), quite a good rough description can be obtained through regular languages. Consider the stylized grammar for the main clause in a Subject-Verb-Object (SVO) language in Figure 5.3. This grammar is quasi-regular: **Verb**, **Adjective** and **Noun** are regular by themselves, **Subject** and **Object** are concatenations of repetitions of regular forms (regular non-terminals and choices) and are therefore quasi-regular, and so is **MainClause**. It takes some work to bring this grammar into standard regular form, but it can be done, as shown in Figure 5.4,

```
MainClause_S  ->  Subject Verb Object
    Subject   ->  [ a | the ] Adjective* Noun
     Object   ->  [ a | the ] Adjective* Noun
       Verb   ->  verb1 | verb2 | ···
  Adjective   ->  adj1 | adj2 | ···
       Noun   ->  noun1 | noun2 | ···
```

Fig. 5.3. A not obviously quasi-regular grammar

in which the lists for verbs, adjectives and nouns have been abbreviated to **verb**, **adjective** and **noun**, to save space.

```
            MainClause_S  ->  a SubjAdjNoun_verb_Object
            MainClause_S  ->  the SubjAdjNoun_verb_Object

SubjAdjNoun_verb_Object   ->  noun verb_Object
SubjAdjNoun_verb_Object   ->  adjective SubjAdjNoun_verb_Object

           verb_Object    ->  verb Object

                Object    ->  a ObjAdjNoun
                Object    ->  the ObjAdjNoun

             ObjAdjNoun   ->  noun
             ObjAdjNoun   ->  adjective ObjAdjNoun

                  verb    ->  verb1 | verb2 | ···
             adjective    ->  adj1 | adj2 | ···
                  noun    ->  noun1 | noun2 | ···
```

Fig. 5.4. A regular grammar in standard form for that of Figure 5.3

Even (finite) context-dependency can be incorporated: for languages that require the verb to agree in number with the subject, we duplicate the first rule:

```
MainClause  ->  SubjectSingular VerbSingular Object
            |   SubjectPlural VerbPlural Object
```

and duplicate the rest of the grammar accordingly. The result is still regular. Nested subordinate clauses may seem a problem, but in practical usage the depth of nesting is severely limited. In English, a sentence containing a subclause containing a subclause containing a subclause will baffle the reader, and even in German and Dutch nestings over say five deep are frowned upon. We replicate the grammar the desired number of times and remove the possibility of further recursion from the deepest level. Then the deepest level is regular, which makes the other levels regular in turn. The resulting grammar will be huge but regular and will be able to profit from all simple and efficient techniques known for regular grammars. The required duplications

and modifications are mechanical and can be done by a program. Dewar, Bratley and Thorne [376] describe an early example of this approach, Blank [382] a more recent one.

5.1.3 Pattern Searching

Many linear patterns, especially text patterns, have a structure that is easily expressed by a (quasi-)regular grammar. Notations that indicate amounts of money in various currencies, for example, have the structure given by the grammar of Figure 5.5, where ␣ has been used to indicate a space symbol. Examples are `$␣19.95` and `¥␣1600`. Such notations, however, do not occur in isolation but are usually embedded in long stretches of text that themselves do not conform to the grammar of Figure 5.5. To

```
        Amount_S  →  CurrencySymbol Space* Digit+ Cents?
CurrencySymbol    →  € | $ | ¥ | £ | ···
         Space    →  ␣
         Digit    →  [0123456789]
         Cents    →  . Digit Digit | .--
```

Fig. 5.5. A quasi-regular grammar for currency notations

isolate the notations, a recognizer (rather than a parser) is derived from the grammar that will accept arbitrary text and will indicate where sequences of symbols are found that conform to the grammar. Parsing (or another form of analysis) is deferred to a later stage. A technique for constructing such a recognizer is given in Section 5.10.

5.1.4 SGML and XML Validation

Finite-state automata also play an important role in the analysis of SGML and XML documents. For the details see Brüggemann-Klein and Wood [150] and Sperberg-McQueen [359], respectively.

5.2 Producing from a Regular Grammar

When producing from a regular grammar, the producer needs to remember only one thing: which non-terminal is next. We shall illustrate this and further concepts using the simple regular grammar of Figure 5.6. This grammar produces sentences consisting of an `a` followed by an alternating sequence of `b`s and `c`s followed by a terminating `a`. For the moment we shall restrict ourselves to regular grammars in standard notation; further on we shall extend our methods to more convenient forms.

The one non-terminal the producer remembers is called its *state* and the producer is said to be *in* that state. When a producer is in a given state, for example `A`, it chooses one of the rules belonging to that state, for example `A→bC`, produces the `b`

```
S_S  →  a A
 S   →  a B
 A   →  b B
 A   →  b C
 B   →  c A
 B   →  c C
 C   →  a
```

Fig. 5.6. Sample regular grammar

and moves to state **C**. Such a move is called a *state transition*, and for a rule $P \rightarrow tQ$ is written $P \xrightarrow{t} Q$. A rule without a non-terminal in the right-hand side, for example **C→a**, corresponds to a state transition to the accepting state; for a rule $P \rightarrow t$ it is written $P \xrightarrow{t} \Diamond$, where $\Diamond$ is the accepting state.

It is customary to combine the states and the possible transitions of a producer in a *transition diagram*. Figure 5.7 shows the transition diagram for the regular grammar of Figure 5.6; we see that, for example, the state transition $\mathbf{A} \xrightarrow{\mathbf{b}} \mathbf{C}$ is represented

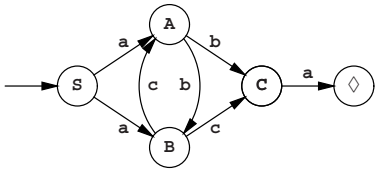


Fig. 5.7. Transition diagram for the regular grammar of Figure 5.6

by the arc marked **b** from **A** to **C**. **S** is the initial state and the accepting state is marked with a $\Diamond$. [1] The symbols on the arcs are those produced by the corresponding move. The producer can stop when it is in an accepting state.

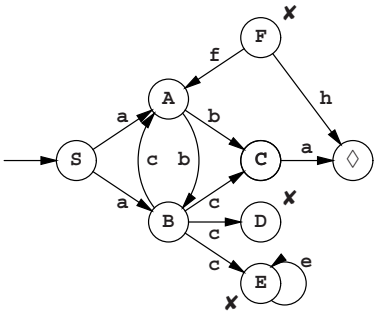
Like the non-deterministic automaton we saw in Section 3.3, the producer is an automaton, or to be more precise, a non-deterministic finite automaton, *NFA* or *finite-state automaton*, *FSA*. It is called "finite" because it can only be in a finite number of states (5 in this case; 3 bits of internal memory would suffice) and "non-deterministic" because, for example, in state **S** it has more than one way to produce an **a**.

Regular grammars can suffer from undefined, unproductive and unreachable non-terminals just like context-free grammars, and the effects are even easier to visualize. If the grammar of Figure 5.6 is extended with the rules

[1] Another convention to mark an accepting state is by drawing an extra circle around it; since we will occasionally want to explicitly mark a non-accepting state, we do not use that convention.

```
B  →  c D   undefined
B  →  c E
E  →  e E   unproductive
F  →  f A   unreachable
F  →  h
```

we obtain the transition diagram



where we can see that no further transitions are defined from **D**, which is the actual meaning of saying that **D** is undefined; that **E**, although being defined, literally has no issue; and that **F** has no incoming arrows.

The same algorithm used for cleaning CF grammars (Section 2.9.5) can be used to clean a regular grammar. Unlike CF grammars, regular grammars and finite-state automata can be minimized: for a given FS automaton *A*, a FS automaton can be constructed that has the least possible number of states and still recognizes the same language as *A*. An algorithm for doing so is given in Section 5.7.

5.3 Parsing with a Regular Grammar

The above automaton for producing a sentence can in principle also be used for parsing. If we have a sentence, for example, **abcba**, and want to check and parse it, we can view the above transition diagram as a maze and the (tokens in the) sentence as a guide. If we manage to follow a path through the maze, matching symbols from our sentence to those on the walls of the corridors as we go, and end up in ◊ exactly at the end of the sentence, we have checked the sentence. See Figure 5.8, where the path is shown as a dotted line. The names of the rooms we have visited form the backbone of the parse tree, which is shown in Figure 5.9.

But finding the correct path is easier said than done. How did we know, for example, to turn left in room **S** rather than right? Of course we could employ general maze-solving techniques (and they would give us our answer in exponential time) but a much simpler and much more efficient answer is available here: we split ourselves in two and head both ways. After the first **a** of **abcba** we are in the set of rooms {**A**, **B**}. Now we have a **b** to follow; from **B** there are no exits marked **b**, but from **A**

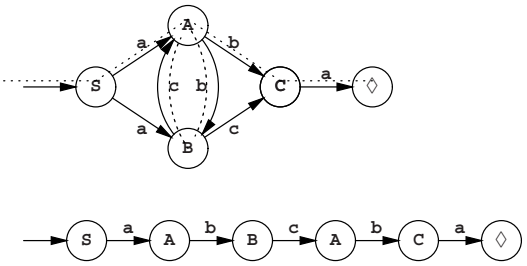


Fig. 5.8. Actual and linearized passage through the maze

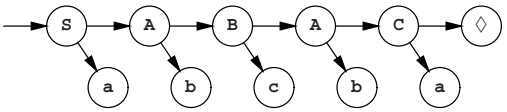


Fig. 5.9. Parse tree from the passage through the maze

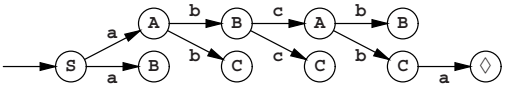


Fig. 5.10. Linearized set-based passage through the maze

there are two, which lead to **B** and **C**. So we are now in rooms {**B C**}. Our path is now more difficult to depict but still easy to linearize, as shown in Figure 5.10.

We can find the parsing by starting at the end and following the pointers backwards: ◊ ⇐ **C** ⇐ **A** ⇐ **B** ⇐ **A** ⇐ **S**. If the grammar is ambiguous the backward pointers may bring us to a fork in the road: an ambiguity has been found and both paths have to be followed separately to find both parsings. With regular grammars, however, one is often not interested in the parse, but only in the recognition: the fact that the input is correct and it ends here suffices.

5.3.1 Replacing Sets by States

Although the process described above is linear in the length of the input (each next token takes an amount of work that is independent of the length of the input), still a lot of work has to be done for each token. What is worse, the grammar has to be consulted repeatedly and so we expect the speed of the process to depend adversely on the size of the grammar. In short, we have designed an interpreter for the non-deterministic automaton, which is convenient and easy to understand, but inefficient.

Fortunately there is a surprising and fundamental improvement possible: from the NFA in Figure 5.7 we construct a new automaton with a new set of states, where each new state is equivalent to a set of old states. Where the original — non-deterministic — automaton was in doubt after the first **a**, a situation we represented as {**A**, **B**}, the new — deterministic — automaton firmly knows that after the first **a** it is in state **AB**.

The states of the new automaton can be constructed systematically as follows. We start with the initial state of the old automaton, which is also the initial state of the new one. For each new state we create, we examine its contents in terms of the old states, and for each token in the language we determine to which set of old states the given set leads. These sets of old states are then considered states of the new automaton. If we create the same state a second time, we do not analyse it again. This process is called the *subset construction* and results initially in a (deterministic) state tree. The state tree for the grammar of Figure 5.6 is depicted in Figure 5.11. To stress

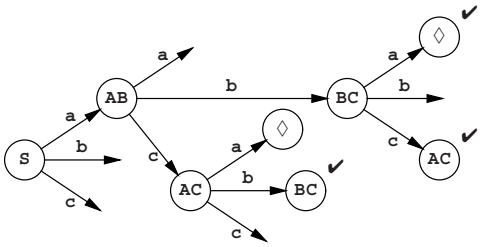


Fig. 5.11. Deterministic state tree for the grammar of Figure 5.6

that it systematically checks all new states for all symbols, outgoing arcs leading nowhere are also shown. Newly generated states that have already been generated before are marked with a ✔.

The state tree of Figure 5.11 is turned into a transition diagram by leading the arrows to states marked ✔ to their first-time representatives and removing the dead ends. The new automaton is shown in Figure 5.12. It is deterministic, and is therefore

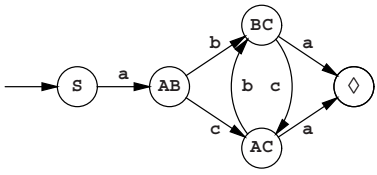


Fig. 5.12. Deterministic automaton for the grammar of Figure 5.6

called a *deterministic finite-state automaton*, or a *DFA* for short.

When we now use the sentence **abcba** as a guide for traversing this transition diagram, we find that we are never in doubt and that we safely arrive at the accepting state. All outgoing arcs from a state bear different symbols, so when following a list of symbols, we are always pointed to at most one direction. If in a given state there is no outgoing arc for a given symbol, then that symbol may not occur in that position. If it does, the input is in error.

There are two things to be noted here. The first is that we see that most of the possible states of the new automaton do not actually materialize: the old automaton had 5 states, so there were $2^5 = 32$ possible states for the new automaton while in fact it has only 5; states like **SB** or **ABC** do not occur. This is usual; although there are non-deterministic finite-state automata with n states that turn into a DFA with 2^n states, these are rare and have to be constructed on purpose. The average garden variety NFA with n states typically results in a DFA with less than or around $10 \times n$ states.

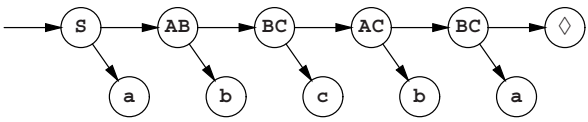
The second is that consulting the grammar is no longer required; the state of the automaton together with the input token fully determine the next state. To allow efficient look-up the next state can be stored in a table indexed by the old state and the input token. The table for our DFA is given in Figure 5.13. Using such a table, an

	input symbol			
	a	**b**	**c**
S	**AB**		
old state **AB**		**BC**	**AC**
AC		◊	**BC**
BC		◊	**AC**

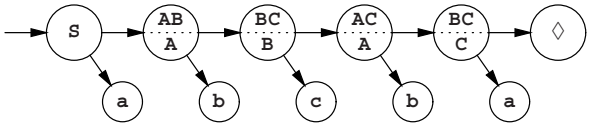
Fig. 5.13. Transition table for the automaton of Figure 5.12

input string can be checked at the cost of only a few machine instructions per token. For the average DFA, most of the entries in the table are empty (cannot be reached by correct input and refer to error states). Since the table can be of considerable size (300 states times 100 tokens is normal), several techniques exist to exploit the empty space by compressing the table. Dencker, Dürre and Heuft [338] give a survey of some techniques.

The parse tree obtained looks as follows:



which is not the original parse tree. If the automaton is used only to recognize the input string this is no drawback. If the parse tree is required, it can be reconstructed in the following fairly obvious bottom-up way. Starting from the last state ◊ and the last token **a**, we conclude that the last right-hand side (the "handle segment" in bottom-up parsing) was **a**. Since the state was **BC**, a combination of **B** and **C**, we look through the rules for **B** and **C**. We find that **a** derived from **C**→**a**, which narrows down **BC** to **C**. The rightmost **b** and the **C** combine into the handle **bC** which in the set {**A**, **C**} must derive from **A**. Working our way backwards we find the parsing



This method again requires the grammar to be consulted repeatedly; moreover, the way back will not always be so straight as in the above example and we will have problems with ambiguous grammars.

Efficient full parsing of regular grammars has received relatively little attention; substantial information can be found in papers by Ostrand, Paull and Weyuker [144] and by Laurikari [151].

5.3.2 ε-Transitions and Non-Standard Notation

A regular grammar in standard form can only have rules of the form $A \rightarrow a$ and $A \rightarrow aB$. We shall now first extend our notation with two other types of rules, $A \rightarrow B$ and $A \rightarrow \varepsilon$, and show how to construct NFAs and DFAs for them. We shall then turn to regular expressions and rules that have regular expressions as right-hand sides (for example, $P \rightarrow a^*bQ$) and show how to convert them into rules in the extended notation.

The grammar in Figure 5.14 contains examples of both new types of rules; Figure

```
S_S  ->  A
 S   ->  a B
 A   ->  a A
 A   ->  ε
 B   ->  b B
 B   ->  b
```

Fig. 5.14. Sample regular grammar with ε-rules

5.15 presents the usual trio of NFA, state tree and DFA for this grammar. First consider the NFA. When we are in state **S** we see the expected transition to state **B** on the token **a**, resulting in the standard rule **S→aB**. The non-standard rule **S→A** indicates that we can get from state **S** to state **A** without reading (or producing) a symbol; we then say that we read the zero-length string ε and that we make an ε-*transition* (or ε-*move*): $\mathbf{S} \xrightarrow{\varepsilon} \mathbf{A}$. The non-standard rule **A→**ε creates an ε-transition to the accepting state: $\mathbf{A} \xrightarrow{\varepsilon} \Diamond$. ε-transitions should not be confused with ε-rules: unit rules create ε-transitions to non-accepting states and ε-rules create ε-transitions to accepting states.

Now that we have constructed an NFA with ε-moves, the question arises how we can process the ε-moves to obtain a DFA. To answer this question we use the same reasoning as before; in Figure 5.7, after having seen an **a** we did not know if we were in state **A** or state **B** and we represented that as {**A**, **B**}. Here, when we enter state **S**, even before having processed a single symbol, we already do not know if we are in

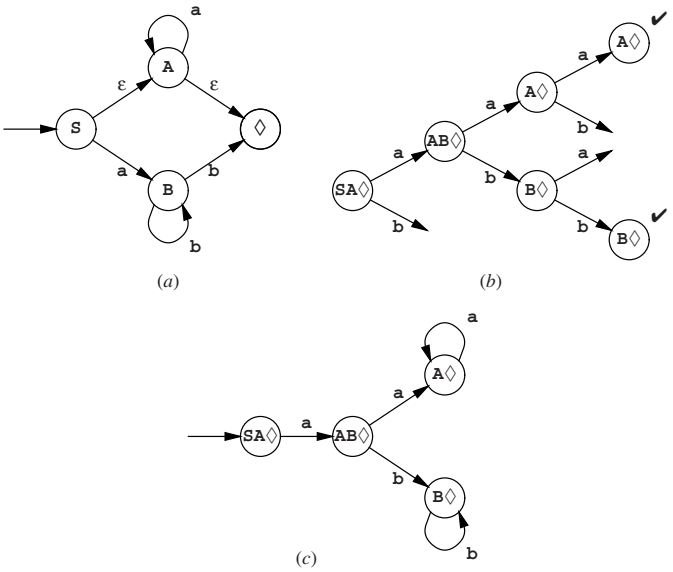


Fig. 5.15. NFA (*a*), state tree (*b*) and DFA (*c*) for the grammar of Figure 5.14

states **S**, **A** or ◊, since the latter two are reachable from **S** through ε-moves. So the initial state of the DFA is already compound: **SA**◊. We now have to consider where this state leads to for the symbols **a** and **b**. If we are in **S** then **a** will bring us to **B** and if we are in **A**, **a** will bring us to **A**. So the new state includes **A** and **B**, and since ◊ is reachable from **A** through ε-moves, it also includes ◊ and its name is **AB**◊. Continuing in this vein we can construct the complete state tree (Figure 5.15(*b*)) and collapse it into a DFA (*c*). Note that all states of the DFA contain the NFA state ◊, so the input may end in all of them.

The set of NFA states reachable from a given state through ε-moves is called the ε-*closure* of that state. The ε-closure of, for example, **S** is {**S**, **A**, ◊}.

For a completely different way of obtaining a DFA from a regular grammar that has recently found application in the field of XML validation, see Brzozowski [139].

5.4 Manipulating Regular Grammars and Regular Expressions

As mentioned in Section 2.3.3, regular languages are often specified by regular expressions rather than by regular grammars. Examples of regular expressions are `[0-9]+(.[0-9]+)?` which should be read as "one or more symbols from the set 0 through 9, possibly followed by a dot which must then be followed by one or more symbols from 0 through 9" (and which represents numbers with possibly a dot in them) and `(ab)*(p|q)+`, which should be read as "zero or more strings **ab** followed by one or more **p**s or **q**s" (and which is not directly meaningful). The usual forms occurring in regular expressions are recalled in the table in Figure 5.16, where

R, R_1, and R_2 are arbitrary regular expressions; some systems provide more forms,

Form	Meaning	Name
R_1R_2	R_1 followed by R_2	concatenation
$R_1 \mid R_2$	R_1 or R_2	alternative
R^*	zero or more Rs	optional sequence (Kleene star)
R^+	one or more Rs	(proper) sequence
$R^?$	zero or one R	optional
(R)	R	grouping
$[abc\cdots]$	any symbol from the set $abc\cdots$	
a	the symbol a itself	

Fig. 5.16. Some usual elements of regular expressions

some provide fewer.

In computer input, no difference is generally made between the metasymbol * and the symbol `*`, etc. Special notations will be necessary if the language to be described contains any of the symbols | `* + ? ( ) [` or `]`.

5.4.1 Regular Grammars from Regular Expressions

A regular expression can be converted into a regular grammar by using the transformations given in Figure 5.17. The T in the transformations stands for an intermediate non-terminal, to be chosen fresh for each application of a transformation; α stands for any regular expression not involving non-terminals, possibly followed by a non-terminal. If α is empty, it should be replaced by ε when it appears alone in a right-hand side.

The expansion from regular expression to regular grammar is useful for obtaining a DFA from a regular expression, as is for example required in lexical analysers like *lex*. The resulting regular grammar corresponds directly to an NFA, which can be used to produce a DFA as described above. There is another method to create an NFA from the regular expression, which requires, however, some preprocessing on the regular expression; see Thompson [140].

We shall illustrate the method using the expression `(ab)*(p|q)+`. Our method will also work for regular grammars that contain regular expressions (like $A \to ab^*cB$) and we shall in fact immediately turn our regular expression into such a grammar:

$$\texttt{S}_\texttt{s} \quad \to \quad \texttt{(ab)*(p|q)+}$$

Although the table in Figure 5.17 uses T for generated non-terminals, we use `A`, `B`, `C`, ... in the example since that is less confusing than $T_1, T_2, T_3, \ldots$. The transformations are to be applied until all rules are in (extended) standard form.

The first transformation that applies is $P \to R^*\alpha$, which replaces $\texttt{S}_\texttt{s} \to \texttt{(ab)*(p|q)+}$ by

Rule pattern	Replace by
$P \to a$	(standard)
$P \to aQ$	(standard)
$P \to Q$	(extended standard)
$P \to \varepsilon$	(extended standard)
$P \to a\alpha$	$P \to aT$
	$T \to \alpha$
$P \to (R_1 \vert R_2 \vert \cdots)\alpha$	$P \to R_1\alpha$
	$P \to R_2\alpha$
	$\cdots$
$P \to (R)\alpha$	$P \to R\alpha$
$P \to R^*\alpha$	$P \to T$
	$T \to RT$
	$T \to \alpha$
$P \to R^+\alpha$	$P \to RT$
	$T \to RT$
	$T \to \alpha$
$P \to R^?\alpha$	$P \to R\alpha$
	$P \to \alpha$
$P \to [abc\cdots]\alpha$	$P \to (a\vert b\vert c\vert \cdots)\alpha$

Fig. 5.17. Transformations on extended regular grammars

```
S_S  ->  A                ✔
 A   ->  (ab) A
 A   ->  (p | q)+
```

The first rule is already in the desired form and has been marked ✔. The transformations $P \to (R)\alpha$ and $P \to a\alpha$ work on `A -> (ab) A` and result in

```
A  ->  a B                ✔
B  ->  b A                ✔
```

Now the transformation $P \to R^+\alpha$ must be applied to `A -> (p | q)+`, yielding

```
A  ->  (p | q)  C
C  ->  (p | q)  C
C  ->  ε                  ✔
```

The ε originated from the fact that `(p | q)+` in `A -> (p | q)+` is not followed by anything (of which ε is a faithful representation). Now `A -> (p | q) C` and `C -> (p | q) C` are easily decomposed into

```
A  ->  p C                ✔
A  ->  q C                ✔
C  ->  p C                ✔
C  ->  q C                ✔
```

```
S_S  →  A
A    →  a B
B    →  b A
A    →  p C
A    →  q C
C    →  p C
C    →  q C
C    →  ε
```

Fig. 5.18. Extended-standard regular grammar for `(ab)*(p|q)+`

The complete extended-standard version can be found in Figure 5.18; an NFA and DFA can now be derived using the methods of Section 5.3.1 (not shown).

5.4.2 Regular Expressions from Regular Grammars

Occasionally, for example in Section 9.12, it is useful to condense a regular grammar into a regular expression. The transformation can be performed by alternatingly substituting a rule and applying the transformation patterns from Figure 5.19. The first

Rule pattern	Replace by
$P \to R_1Q_1$ $P \to R_2Q_2$ $\cdots$	$P \to R_1Q_1 \mid R_2Q_2 \cdots$
$P \to R_1Q \mid R_2Q \mid \cdots Q \mid \alpha$	$P \to (R_1 \mid R_2 \mid \cdots)Q \mid \alpha$
$P \to (R)P \mid R_1Q_1 \mid R_2Q_2 \mid \alpha$	$P \to (R)^*R_1Q_1 \mid (R)^*R_2Q_2 \mid \beta$

Fig. 5.19. Condensing transformations on regular grammars

pattern combines all rules for the same non-terminal. The second pattern combines all regular expressions that precede the same non-terminal in a right-hand side; α is a list of alternatives that do not end in Q (but see next paragraph). The third pattern removes right recursion: if the repetitive part is (R), it prepends $(R)^*$ to all non-recursive alternatives; here β consists of all the alternatives in α, with $(R)^*$ prepended to each of them. $Q_1, Q_2, \cdots$ should not be equal to P (but see next paragraph). When α is ε it can be left out when it is concatenated with a non-empty regular expression.

The substitutions and transformations may be applied in any order and will always lead to a correct regular expression, but the result depends heavily on the application order; to obtain a "nice" regular expression, human guidance is needed. Also, the two conditions in the previous paragraph may be violated without endangering the correctness, but the result will be a more "ugly" regular expression.

We will now apply the transformation to the regular grammar of Figure 5.18, and will not hesitate to supply the human guidance. We first combine the rules by their left-hand sides (transformation 1):

```
S  →  A
A  →  a B | p C | q C
B  →  b A
C  →  p C | q C | ε
```

Next we substitute **B**:

```
A  →  a b A | p C | q C
C  →  p C | q C | ε
```

followed by scooping up prefixes (transformation 2):

```
A  →  (ab) A | (p|q) C
C  →  (p|q) C | ε
```

Note that we have also packed the **ab** that prefixes **A**, to prepare it for the next transformation, which involves turning recursion into repetition:

```
S  →  A
A  →  (ab)* (p|q) C
C  →  (p|q)*
```

Now **C** can be substituted in **A** and **A** in **S**, resulting in

```
S  →  (ab)* (p|q) (p|q)*
```

This is equivalent but not identical to the **(ab)*(p|q)+** we started with.

5.5 Manipulating Regular Languages

In Section 2.10 we discussed the set operations "union", "intersection", and "negation" on CF languages, and saw that the latter two do not always yield CF languages. For regular languages the situation is simpler: these set operations on regular languages always yield regular languages.

Creating a FS automaton for the union of two regular languages defined by the FS automata A_1 and A_2 is trivial: just create a new start state and add ε-transitions from that state to the start states of A_1 and A_2. If need be the ε-transitions can then be removed as described in Section 5.3.1.

There is an interesting way to get the negation (complement) of a regular language L defined by a FS automaton, provided the automaton is ε-free. When an automaton is ε-free, each state t in it shows directly the set of tokens C_t with which an input string that brings the automaton in state t can continue: C_t is exactly the set of tokens for which t has an outgoing transition. This means that if the string continues with a token which is not in C_t, the string is not in L, and so we may conclude it is in $\neg L$. Now we can "complete" state t by adding outgoing arrows on all tokens not in C_t and lead these to a non-accepting state, which we will call s_{-1}. If we perform this completion for all states in the automaton, including s_{-1}, we obtain a so-called *complete automaton*, an automaton in which all transitions are defined.

The complete version of the automaton of Figure 5.7 is shown in Figure 5.20, where the non-accepting state is marked with a ✘.

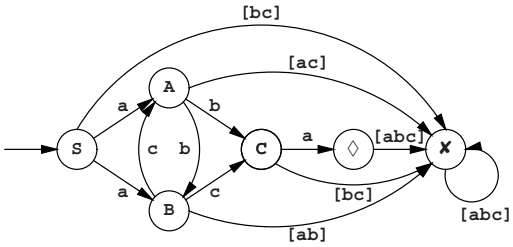


Fig. 5.20. The automaton of Figure 5.7 completed

The importance of a complete automaton lies in the fact that it never gets stuck on any (finite) input string. For those strings that belong to the language L of the automaton, it ends in an accepting state; for those that do not it ends in a non-accepting state. And this immediately suggests how to get an automaton for the complement (negative) of L: swap the status of accepting and non-accepting states, by making the accepting states non-accepting and the non-accepting states accepting!

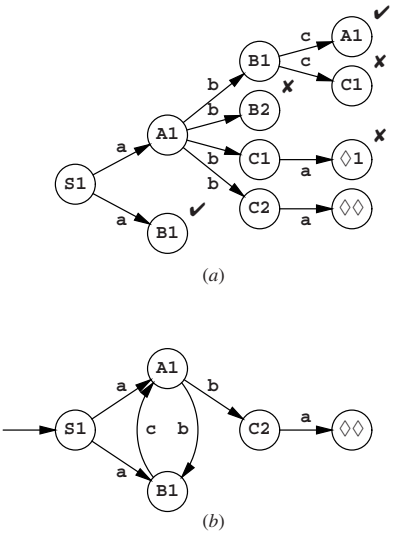
Note that completing the automaton has damaged its error detection properties, in that it will not reject an input string at the first offending character but will process the entire string and only then give its verdict.

The completion process requires the automaton to be ε-free. This is easily achieved by making it deterministic, as described on page 145, but that may be overkill. See Problem 5.4 for a way to remove the ε-transitions only.

Now that we have negation of FSAs, constructing the intersection of two FSAs seems easy: just negate both automata, take the union, and negate the result, in an application of De Morgan's Law $p \cap q = \neg((\neg p) \cup (\neg q))$. But there is a hitch here. Constructing the negation of an FSA is easy only if the automaton is ε-free, and the union in the process causes two ε-transitions in awkward positions, making this "easy" approach quite unattractive.

Fortunately there is a simple trick to construct the intersection of two FS automata that avoids these problems: run both automata simultaneously, keeping track of their two states in one single new state. As an example we will intersect automaton A_1, the automaton of Figure 5.7, with an FSA A_2 which requires the input to contain the sequence **ba**. A_2 is represented by the regular expression $\texttt{.}^*\texttt{ba.}^*$. It needs 3 states, which we will call **1** (start state), **2** and ◊ (accepting state); it has the following transitions: $\texttt{1} \xrightarrow{\texttt{[abc]}} \texttt{1}$, $\texttt{1} \xrightarrow{\texttt{b}} \texttt{2}$, $\texttt{2} \xrightarrow{\texttt{a}} \lozenge$, $\lozenge \xrightarrow{\texttt{[abc]}} \lozenge$.

We start the intersection automaton $A_1 \cap A_2$ in the combined state **S1**, which is composed of the start state **S** of A_1 and the start state **1** of A_2. For each transition $P_1 \xrightarrow{t} Q_1$ in A_1 and for each transition $P_2 \xrightarrow{t} Q_2$ in A_2 we create a transition $(P_1P_2) \xrightarrow{t} (Q_1Q_2)$ in $A_1 \cap A_2$. This leads to the state tree in Figure 5.21(*a*); the corresponding FSA is in (*b*). We see that it is similar to that in Figure 5.7, except that the transition $\texttt{B} \xrightarrow{\texttt{C}} \texttt{C}$ is missing: the requirement that the string should contain the sequence **ba** removed it.



(*a*)

(*b*)

Fig. 5.21. State tree (*a*) and FSA (*b*) of the intersection of Figure 5.7 and `.*ba.*`

In principle, the intersection of an FSA with n states and one with m states can require $n \times m$ states, but in practice something like $c \times (n+m)$ for some small value of c is more usual.

Conversely, sometimes a complex FSA can be decomposed into the intersection of two much simpler FSAs, with great gain in memory requirements, and sometimes it cannot. There is unfortunately little theory on how to do this, though there are some heuristics; see Problem 5.7. The process is also called "factorization", but that is an unfortunate term, since it suggests the same uniqueness of factorization we find in integers, and the decomposition of FSAs is not unique.

5.6 Left-Regular Grammars

In a left-regular grammar, all rules are of the form $A \rightarrow a$ or $A \rightarrow Ba$ where a is a terminal and A and B are non-terminals. Figure 5.22 gives a left-regular grammar equivalent to that of Figure 5.6.

Left-regular grammars are often brushed aside as just a variant of right-regular grammars, but their look and feel is completely different. Take the process of producing a string from this grammar, for example. Suppose we want to produce the sentence **abcba** used in Section 5.3. To do so we have to first decide all the states we are going to visit, and only when the last one has been decided upon can the first token be produced:

```
S_S  →  C a
C    →  B c
C    →  A b
B    →  A b
B    →  a
A    →  B c
A    →  a
```

Fig. 5.22. A left-regular grammar equivalent to that of Figure 5.6

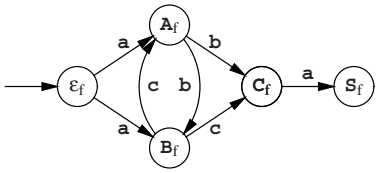
```
S
C a
A b a
B c b a
A b c b a
a b c b a
```

And once the first token is available, all of them are, and we do not have any choice any more; this is vastly different from producing from a right-regular grammar.

Parsing with a left-regular grammar is equally weird. It is easy to see that initially we are in a union of all states {**S**,**A**,**B**,**C**}, but if we now see an **a** in the input, we can move over this **a** in two rules, **B→a**, and **A→a**. Suppose we use rule **A→a**; what state are we in now? The rule specifies no state except **A**; so what does the move mean?

The easy way out is to convert the grammar to a right-regular one (see below in this section), but it is more interesting to try to answer the question what a move over **a** in **A→a** means. The only thing we know after such a move is that we have just completed a production of **A**, so the state we are in can justifiably be described as "**A** finished"; we will write such a state as $\mathbf{A}_\mathrm{f}$. And in the same manner the first rule in Figure 5.22 means that when we are in a state $\mathbf{C}_\mathrm{f}$ and we move over an **a** we are in a state $\mathbf{S}_\mathrm{f}$; this corresponds to a transition $\mathbf{C}_\mathrm{f} \xrightarrow{\mathbf{a}} \mathbf{S}_\mathrm{f}$. Then we realize that "S finished" means that we have parsed a complete terminal production of **S**; so the state $\mathbf{S}_\mathrm{f}$ is the accepting state $\Diamond$ and we see the rightmost transition in Figure 5.7 appear.

Now that we have seen that the rule $A \rightarrow Bt$ corresponds to the transition $B_f \xrightarrow{t} A_f$, and that the rule $S_S \rightarrow Bt$ corresponds to $B_f \xrightarrow{t} \Diamond$, what about rules of the form $A \rightarrow t$? After the transition over t we are certainly in the state A_f, but where did we start from? The answer is that we have not seen any terminal production yet, so we are in a state ε_f, the start state! So the rules **A→a** and **B→a** correspond to transitions $\varepsilon_\mathrm{f} \xrightarrow{\mathbf{a}} \mathbf{A}_\mathrm{f}$ and $\varepsilon_\mathrm{f} \xrightarrow{\mathbf{a}} \mathbf{B}_\mathrm{f}$, two more components of Figure 5.7. Continuing this way we quickly reconstruct the transition diagram of Figure 5.7, with modified state names:



This exposes an awkward asymmetry between start state and accepting state, in that unlike the start state the accepting state corresponds to a symbol in the grammar. This asymmetry can be partially removed by representing the start state by a more neutral symbol, for example □. We then obtain the following correspondence between our right-regular and left-regular grammar:

```
□ → a A        A → □ a
□ → a B        B → □ a
A → b B        B → A b
A → b C        C → A b
B → c A        A → B c
B → c C        C → B c
C → a ◊        ◊ → C a

□: start state  □: ε
◊: ε            ◊: start state
```

Obtaining a regular expression from a left-regular grammar is simple: most of the algorithm in Section 5.4.2 can be taken over with minimal change. Only the transformation that converts recursion into repetition

Rule pattern	Replace by
$P \rightarrow (R)P \mid R_1Q_1 \mid R_2Q_2 \mid \alpha$	$P \rightarrow (R)^*R_1Q_1 \mid (R)^*R_2Q_2 \mid \beta$

must be replaced by

$$P \rightarrow P(R) \mid Q_1R_1 \mid Q_2R_2 \mid \alpha \qquad P \rightarrow Q_1R_1(R)^* \mid Q_2R_2(R)^* \mid \beta'$$

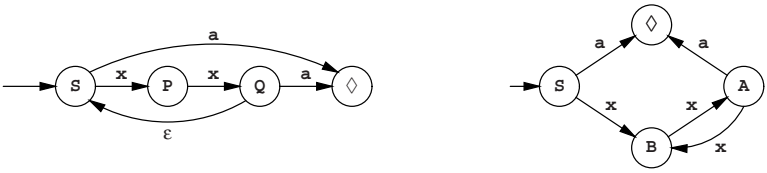
where β' consists of all the alternatives in α, with $(R)^*$ appended to each of them. This is because **A→aA|b** yields $\mathtt{a}^*\mathtt{b}$ but **A→Aa|b** yields $\mathtt{ba}^*$.

5.7 Minimizing Finite-State Automata

Turning an NFA into a DFA usually increases the size of the automaton by a moderate factor, perhaps 10 or so, and may occasionally grossly inflate the automaton. Considering that for a large automaton a size increase of a factor of say 10 can pose a major problem; that even for a small table any increase in size is undesirable if the table has to be stored in a small electronic device; and that large inflation factors may occur unexpectedly, it is often worthwhile to try to reduce the number of states in the DFA.

The key idea of the *DFA minimization algorithm* presented here is that we consider states to be equivalent until we can see a difference. To this end the algorithm keeps the DFA states in a number of mutually disjoint subsets, a "partition." A *partition* of a set S is a collection of subsets of S such that each member of S is in exactly one of those subsets; that is, the subsets have no elements in common and their union is the set S. The algorithm iteratively splits each subset in the partition as long as it can see a difference between states in it.

We will use the DFA from Figure 5.23(b) as an example; it can be derived from the NFA in Figure 5.23(a) through the subset algorithm with **A** = **SQ** and **B** = **P**, and is not minimal, as we shall see.



(a) (b)

Fig. 5.23. A non-deterministic FSA and the resulting deterministic but not minimal FSA

Initially we partition the set of states into two subsets: one containing all the accepting states, the other containing all the other states; these are certainly different. In our example this results in one subset containing states **S**, **B** and **A**, and one subset containing the accepting state ◊.

Next, we process each subset S_i in turn. If there exist two states q_1 and q_2 in S_i that on some symbol a have transitions to members of different subsets in the current partition, we have found a difference and S_i must be split. Suppose we have $q_1 \xrightarrow{a} r_1$ and $q_2 \xrightarrow{a} r_2$, and r_1 is in subset X_1 and r_2 is in a different subset X_2, then S_i must be split into one subset containing q_1 and all other states q_j in S_i which have $q_j \xrightarrow{a} r_j$ with r_j in X_1, and a second subset containing the other states from S_i. If q_1 has no transition on a but q_2 does, or vice versa, we have also found a difference and S_i must be split as well.

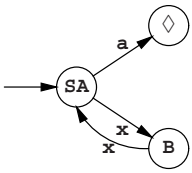
In our example, states **S** and **A** have transitions on **a** (to the same state, ◊), but state **B** does not, so this step results in two subsets, one containing the states **S** and **A**, and the other containing state **B**.

We repeat applying this step to all subsets in the partition, until no subset can be split any more. This will eventually happen, because the total number of subsets is bounded: there can be no more subsets in a partition than there are states in the original DFA, and during the process subsets are never merged. (This is another example of a closure algorithm.)

When this process is completed, all states in a subset S_i of the resulting partition have the property that for any alphabet symbol a their transition on a ends up in the same subset $S_i(a)$ of the partition. Therefore, we can consider each subset to be a sin-

gle state in the *minimized DFA*. The start state of the minimized DFA is represented by the subset containing the start state of the original DFA, and the accepting states of the minimized DFA are represented by the subsets containing accepting states of the original DFA. The resulting DFA is, in fact, the smallest DFA that recognizes the language specified by the DFA that we started with. See, for example, Hopcroft and Ullman [391].

In our example we find no further splits, and the resulting DFA is depicted below.



5.8 Top-Down Regular Expression Recognition

The Type 3 recognition technique of Section 5.3 is a bottom-up method collecting hypotheses about the reconstruction of the production process, with a top-down component making sure that the recognized string derives from the start symbol. In fact, the subset algorithm can be derived quite easily from a specific bottom-up parser, the Earley parser, which we will meet in Section 7.2 (Problem 5.9). Somewhat surprisingly, much software featuring regular expressions uses the straightforward backtracking top-down parser from Section 6.6, adapted to regular expressions. The main advantage is that this method does not require preprocessing of the regular expression; the disadvantage is that it may require much more than linear time. We will first explain the technique briefly (backtracking top-down parsing is more fully discussed in Section 6.6), and then return to the advantages and disadvantages.

5.8.1 The Recognizer

The top-down recognizer follows the grammar of regular expressions, which we summarize here:

```
regular_expression_S  ->  compound_re*
        compound_re   ->  repeat_re | simple_re
          repeat_re   ->  simple_re ['*'|'+'|'?']
          simple_re   ->  token | '(' regular_expression ')'
```

The recognizer keeps two pointers, one in the regular expression and one in the input, and tries to move both in unison: when a token is matched both pointers move one position forward, but when a **`simple_re`** must be repeated, the regular expression pointer jumps backwards, and the input pointer stays in place. When the regular expression pointer points to the end of the regular expression, the recognizer registers a match, based on how far the input pointer got.

When the recognizer tries to recognize a **compound_re**, it first finds out whether it is a **repeat_re**. If so, it checks the mark. If that is a **+** indicating a mandatory **simple_re**, the recognizer just continues searching for a **simple_re**, but if the **simple_re** is optional (*****, **?**), the search splits in two: one for a **simple_re**, and one for the rest of the regular expression, after this **repeat_re**. When the recognizer comes to the end of a **repeat_re**, it again checks the mark. If it is a **?**, it just continues, but if it was a real repeater (*****, **+**), the search again splits in two: one jumping back to the beginning of the **repeat_re**, and one continuing with the rest of the regular expression.

When the recognizer finds that the **simple_re** is a **token**, it compares the token with the token at the input pointer. If they match, both pointers are advanced; otherwise this search is abandoned.

Two questions remain: how do we implement the splitting of searches, and what do we do with the recorded matches. We implement the search splitting by doing them sequentially: we first do the entire first search up to the end or failure, including all its subsearches; then, regardless of the result, we do the second search. This sounds bothersome, both in coding and in efficiency, but it isn't. The skeleton code for the optional **repeat_re** is just

```
procedure try_optional_repeat_re(rp, ip: int):
begin
    try_simple_re(rp, ip);
    try_regular_expression(after_subexpression(rp), ip);
end;
```

where `rp` and `ip` are the regular expression pointer and the input pointer. And the algorithm is usually quite efficient, since almost all searches fail immediately because a **token** search compares two non-matching tokens.

The processing of the recorded matches depends on the application. If we want to know if the regular expression matches the entire string, as for example in file name matching, we check if we have simultaneously reached the end of the input, and if so, we abandon all further searches and return success; if not, we just continue searching. But if, for example, we want the longest match, we keep a high-water mark and continue until all searches have been exhausted.

5.8.2 Evaluation

Some advantages of top-down regular expression matching are obvious: the algorithm is very easy to program and involves no or hardly any preprocessing of the regular expression, depending on the implementation of structuring routines like **after_subexpression()**. Other advantages are less directly visible. For example, the technique allows naming a part of the regular expression and checking its repeated presence somewhere else in the input; this is an unexpectedly powerful feature. A simple example is the pattern **(.*)=x\x**, which says: match an arbitrary segment of the input, call it **x**, and then match the rest of the input to whatever has been recognized for **x**; **\x** is called a *backreference*. (A more usual but less clear

notation for the same regular expression is `\(.*\)\1`, in which `\1` means: match the first subexpression enclosed in `\(` and `\)`.)

Faced with the input **abab**, the recognizer sets **x** to the values ε, **a**, **ab**, **aba**, and **abab** in any order, and then tries to match the tail left over in each case to the present value of **x**. This succeeds only for **x**=ε and **x**=**ab**, and only in the last case is the whole input recognized. So the above expression recognizes the language of all strings that consist of two identical parts: *ww*, where *w* is any string over the given alphabet. Since this is a context-sensitive language, we see to our amazement that, skipping the entire Type 2 languages, the Type 3 regular expressions with backreferences recognize a Type 1 language! A system which uses this feature extensively is the §-calculus (Jackson [285, 291]), discussed further in Section 15.8.3.

The main disadvantage of top-down regular expression recognition is its time requirements. Although they are usually linear with a very modest multiplication constant, they can occasionally be disturbingly high, especially at unexpected moments. $O(n^k)$ time requirement occur with patterns like $a^*a^*\cdots a^*$, where the a^* is repeated k times, so in principle the cost can be any polynomial in the length of the input, but behavior worse than quadratic is unusual. Finding all 10000 occurrences of lines matching the expression `.*)` in this book took 36 sec.; finding all 11000 occurrences of just the `)` took no measurable time.

5.9 Semantics in FS Systems

In FS systems, semantic actions can be attached to states or to transitions. If the semantics is attached to the states, it is available all the time and is static. It could control an indicator on some panel of some equipment, or keep the motor of an elevator running. Semantics associated with the states is also called *Moore semantics* (Moore [136]).

If the semantics is attached to the transitions, it is available only at the moment the transition is made, in the form of a signal or procedure call; it is dynamic and transitory. Such a signal could cause a plastic cup to drop in a coffee machine or shift railroad points; the stability, staticness, is then provided by the physical construction of the equipment. And a procedure call could tell the lexical analyser in a compiler that a token **begin** has been found. Semantics associated with transitions is also called *Mealy semantics* (Mealy [134]).

There are many variants of transition-associated semantics. The signal can come when specific transition $s_i \xrightarrow{t} s_j$ occurs (Mealy [134]); when a specific token causes a specific state to be entered ($* \xrightarrow{t} s_j$, where $*$ is any state); when a specific state is entered ($* \xrightarrow{*} s_j$, McNaughton and Yamada [137]); when a specific state is left ($s_j \xrightarrow{*} *$); etc. Not much has been written about these differences. Upon reading a paper it is essential to find out which convention the author(s) use. In practical situations it is usually self-evident which variant is the most appropriate.

5.10 Fast Text Search Using Finite-State Automata

Suppose we are looking for the occurrence of a short piece of text, for example, a word or a name (the "search string") in a large piece of text, for example, a dictionary or an encyclopedia. One naive way of finding a search string of length n in a text would be to try to match it to the characters 1 to n; if that fails, shift the pattern one position and try to match against characters 2 to $n+1$, etc., until we find the search string or reach the end of the text. This process is, however, costly, since we may have to look at each character n times.

Finite automata offer a much more efficient way to do text search. We derive a DFA from the string, let it run down the text and when it reaches an accepting state, it has found the string. Assume for example that the search string is **ababc** and that the text will contain only **a**s, **b**s and **c**s. The NFA that searches for this string is shown in Figure 5.24(a); it was derived as follows. At each character in the text there are two

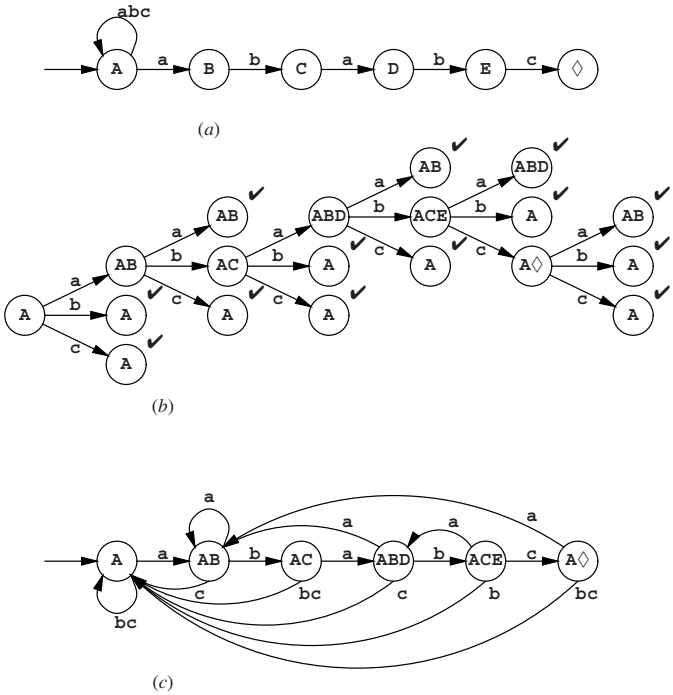


Fig. 5.24. NFA (a), state tree (b) and DFA (c) to search for **ababc**

possibilities: either the search string starts there, which is represented by the chain of states going to the right, or it does not start there, in which case we have to skip the present character and return to the initial state. The automaton is non-deterministic, since when we see an **a** in state A, we have two options: to believe that it is the start of an occurrence of **ababc** or not to believe it.

Using the traditional techniques, this NFA can be used to produce a state tree (*b*) and then a DFA (*c*). Figure 5.25 shows the states the DFA goes through when fed the text **aababababca**. We see that we have implemented *superstring recognition*, in

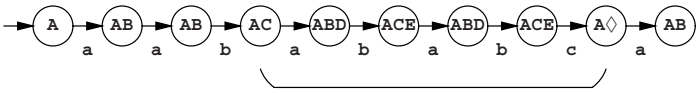


Fig. 5.25. State transitions of the DFA of Figure 5.24(*c*) on **aababababca**

which a substring of the input is recognized as matching the grammar rather than the entire input. This makes the input a superstring of a string in the language, hence the name.

This application of finite-state automata is known as the *Aho and Corasick bibliographic search algorithm* (Aho and Corasick [141]). Like any DFA, it requires only a few machine instructions per character. As an additional bonus it will search for several strings for the price of one. The DFA corresponding to the NFA of Figure 5.26 will search simultaneously for **Kawabata**, **Mishima** and **Tanizaki**. Note

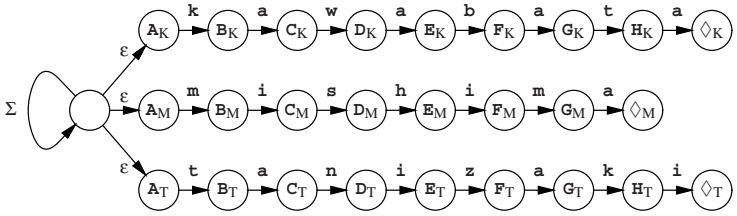


Fig. 5.26. Example of an NFA for searching multiple strings

that three different accepting states result, $\Diamond_K$, $\Diamond_M$ and $\Diamond_T$.

The Aho and Corasick algorithm is not the last word in string search. It faces stiff competition from the Rabin-Karp algorithm (Karp and Rabin [145]) and the Boyer-Moore algorithm (Boyer and Moore [143]). An excellent overview of fast string search algorithms is given by Aho [147]. Watson [149] extends the Boyer-Moore technique, which searches for a single word, so it can search for a regular expression. However fascinating all these algorithms are, they are outside the scope of this book and will not be treated here.

5.11 Conclusion

Regular grammars are characterized by the fact that no nesting is involved. Switching from one grammar rule or transition network to another is a memory-less move.

Consequently the production process is determined by a single position in the grammar and the recognition process is determined by a finite number of positions in the grammar.

Regular grammars correspond to regular expression, and vice versa, although the conversion algorithms tend to produce results that are more complicated than would be possible.

Strings in a regular set can be recognized bottom-up, using finite-state automata created by the "subset algorithm", or top-down, using recursive descent routines derived from the regular expression. The first has the advantage that it is very efficient; the second allows easy addition of useful semantic actions and recognition restrictions.

Finite-state automata are extremely important in all kinds of text searches, from bibliographical and Web searches through data mining to virus scanning.

Problems

Problem 5.1: Construct the regular grammar for open parts of sentential forms in rightmost derivations for the grammar C in Section 5.1.1.

Problem 5.2: The FS automata in Figures 5.7 and 5.12 have only one accepting state, but the automaton in Figure 5.15(c) has several. Are multiple accepting states necessary? In particular: 1. Can any FS automaton A be transformed into an equivalent single accepting state FS automaton B? 2. So that in addition B has no ε-transitions? 3. So that in addition B is deterministic?

Problem 5.3: Show that the grammar cleaning operations of removing non-productive rules and removing unreachable non-terminals can be performed in either order when cleaning a regular grammar.

Problem 5.4: Design an algorithm for removing ε-transitions from a FS automaton.

Problem 5.5: Design a way to perform the completion and negation of a regular automaton (Section 5.5) on the regular grammar rather than on the automaton.

Problem 5.6: *For readers with a background in logic:* Taking the complement of the complement of an FSA does not always yield the original automaton, but taking the complement of the complement of an already complemented FSA does, which shows that complemented automata are in some way different. Analyse this phenomenon and draw parallels with intuitionistic logic.

Problem 5.7: *Project:* Study the factorization/decomposition of FSAs; see, for example, Roche, [148].

Problem 5.8: When we assign *two* states to each non-terminal A, A_s for "A start" and A_f for "A finished, a rule $A \rightarrow XY$ results in 3 ε-transitions, $A_s \xrightarrow{\varepsilon} X_s$, $X_f \xrightarrow{\varepsilon} Y_s$ and $Y_f \xrightarrow{\varepsilon} A_f$, and a non-ε-transition $X_s \xrightarrow{X} X_f$ or $Y_s \xrightarrow{Y} Y_f$, depending on whether X or Y is a terminal. Use this view to write a more symmetrical and esthetic account of left- and right-regular grammars than given in Section 5.6.

Problem 5.9: Derive the subset algorithm from the Earley parser (Section 7.2) working on a left-regular grammar.

Problem 5.10: Derive a regular expression for `S` from the grammar of Figure 5.22.

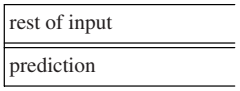
Problem 5.11: *Project:* Section 5.7 shows how to minimize a FS automaton/grammar by initially assuming all non-terminal are equal. Can a CF grammar be subjected to a similar process and what will happen?

Problem 5.12: *History:* Trace the origin of the use of the Kleene star, the raised star meaning "the set of an unbounded number of occurrences". (See [135].)

6

General Directional Top-Down Parsing

In this chapter, we will discuss top-down parsing methods that try to rederive the input sentence by prediction. As explained in Section 3.2.1, we start with the start symbol and try to produce the input from it; at any point in time, we have a sentential form that represents our prediction of the rest of the input sentence. It is convenient to draw the prediction right under the part of the input that it predicts, with their left ends flush, as we did in Figure 3.5:



This sentential form consists of both terminals and non-terminals. If a terminal symbol is in front, we match it with the current input symbol. If a non-terminal is in front, we pick one of its right-hand sides and replace the non-terminal with this right-hand side. This way, we all the time replace the leftmost non-terminal, and in the end, if we succeed, we have imitated a leftmost derivation. Note that the prediction part corresponds to the open part of the sentential form when doing leftmost derivation, as discussed in Section 5.1.1.

6.1 Imitating Leftmost Derivations

Let us now illustrate such a rederiving process with an example. Consider the grammar of Figure 6.1. This grammar produces all sentences with equal numbers of **a**s

```
S_S  ->  aB | bA
 A   ->  a  | aS | bAA
 B   ->  b  | bS | aBB
```

Fig. 6.1. A grammar producing all sentences with equal numbers of **a**s and **b**s

and **b**s.

Let us try to parse the sentence **aabb**, by trying to rederive it from the start symbol, **S**. **S** is our first prediction. The first symbol of our prediction is a non-terminal, so we have to replace it by one of its right-hand sides. In this grammar, there are two choices for **S**: either we use the rule **S→aB**, or we use the rule **S→bA**. The sentence starts with an **a** and not with a **b**, so we cannot use the second rule here. Applying the first rule leaves us with the prediction **aB**. Now the first symbol of the prediction is a terminal symbol. Here, we have no choice:

a	abb
a	B

We have to match this symbol with the current symbol of the sentence, which is also an **a**. So we have a match, and accept the **a**. This leaves us with the prediction **B** for the rest of the sentence: **abb**. The first symbol of the prediction is again a non-terminal, so it has to be replaced by one of its right-hand sides. Now we have three choices. However, the first and the second are not applicable here, because they start with a **b**, and we need another **a**. Therefore, we take the third choice, so now we have prediction **aBB**:

a	a	bb
a	a	BB

Again, we have a match with the current input symbol, so we accept it and continue with the prediction **BB** for **bb**. Again, we have to replace the leftmost **B** by one of its choices. The next terminal in the sentence is a **b**, so the third choice is not applicable here. This still leaves us with two choices, **b** and **bS**. So, we can either try them both, or be a bit more intelligent about it. If we would take **bS**, then we would get at least another **a** (because of the **S**), so this cannot be the right choice. So we take the **b** choice, and get the prediction **bB** for **bb**. Again, we have a match, and this leaves us with prediction **B** for **b**. For the same reason, we take the **b** choice again. After matching, this leaves us with an empty prediction. Luckily, we are also at the end of the input sentence, so we accept it. If we had made notes of the production rules used, we would have found the following derivation:

S → aB → aaBB → aabB → aabb

Figure 6.2 presents the steps of the parse in a tree-form. The dashed line separates the already processed part from the prediction. All the time, the leftmost symbol of the prediction is processed.

This example demonstrates several aspects that the parsers discussed in this chapter have in common:

- We always process the leftmost symbol of the prediction.
- If this symbol is a terminal, we have no choice: we have to match it with the current input symbol or reject the parse.

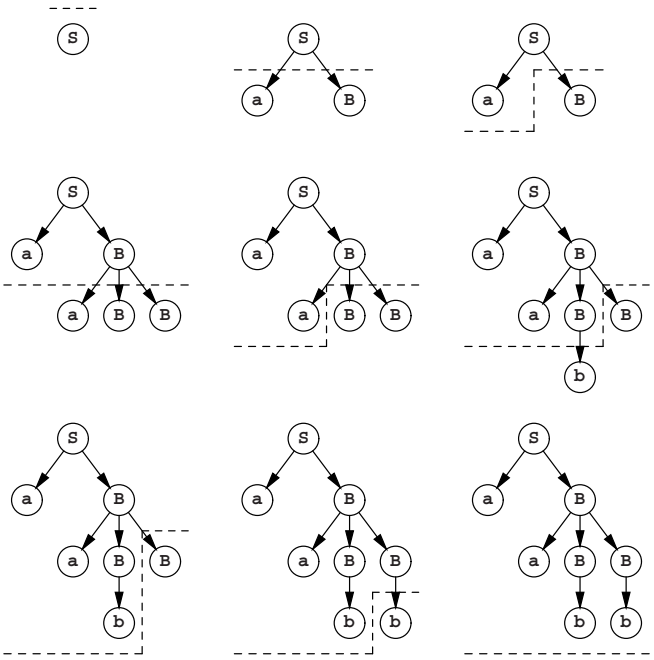


Fig. 6.2. Production trees for the sentence **aabb**

- If this symbol is a non-terminal, we have to make a prediction: it has to be replaced by one of its right-hand sides. Thus, we always process the leftmost non-terminal first, so we get a leftmost derivation.
- As a result, the top-down method recognizes the nodes of the parse tree in pre-order: the parent is identified before any of its children.

6.2 The Pushdown Automaton

The steps we have taken in the example above resemble very much the steps of a so-called *pushdown automaton*. A pushdown automaton (*PDA*) is an imaginary mathematical device that reads input and has control over a stack. The stack can contain symbols that belong to a so-called *stack alphabet*. A *stack* is a list that can only be accessed at one end: the last symbol entered on the list ("pushed") is the first symbol to be taken from it ("popped"). This is also sometimes called a "first-in, last-out" list, or a *FILO list*: the first symbol that goes in is the last symbol to come out. In the example above, the prediction works like a stack, and this is what the pushdown automaton uses the stack for too. We therefore call this stack the *prediction stack*. The stack also explains the name "pushdown" automaton: the automaton "pushes" symbols on the stack for later processing.

The pushdown automaton operates by popping a stack symbol and reading an input symbol. These two symbols then in general give us a choice of several lists

of stack symbols to be pushed on the stack. So there is a mapping of (input symbol, stack symbol) pairs to lists of stack symbols. The automaton accepts the input sentence when the stack is empty at the end of the input. If there are choices (so an (input symbol, stack symbol) pair maps to more than one list), the automaton accepts a sentence when there are choices that lead to an empty stack at the end of the sentence.

This automaton is modeled after context-free grammars with rules in the so-called *Greibach Normal Form* (*GNF*). In this normal form, all grammar rules have either the form $A \rightarrow a$ or $A \rightarrow aB_1B_2 \cdots B_n$, with a a terminal and $A, B_1, \ldots, B_n$ non-terminals. The stack symbols are, of course, the non-terminals. A rule of the form $A \rightarrow aB_1B_2 \cdots B_n$ leads to a mapping of the (a, A) pair to the list $B_1B_2 \cdots B_n$. This means that if the input symbol is an a, and the prediction stack starts with an A, we could accept the a, and replace the A part of the prediction stack with $B_1B_2 \cdots B_n$. A rule of the form $A \rightarrow a$ leads to a mapping of the (a, A) pair to an empty list. The automaton starts with the start symbol of the grammar on the stack. Any context-free grammar that does not produce the empty string can be put into Greibach Normal Form (Greibach [8]). Most books on formal language theory discuss how to do this (see for example Hopcroft and Ullman [391]).

The example grammar of Figure 6.1 already is in Greibach Normal Form, so we can easily build a pushdown automaton for it. The automaton is characterized by the mapping shown in Figure 6.3.

```
(a, S)  ->  B
(b, S)  ->  A
(a, A)  ->
(a, A)  ->  S
(b, A)  ->  AA
(b, B)  ->
(b, B)  ->  S
(a, B)  ->  BB
```

Fig. 6.3. Mapping of the PDA for the grammar of Figure 6.1

An important remark to be made here is that many pushdown automata are non-deterministic. For example, the pushdown automaton of Figure 6.3 can choose between an empty list and an `S` for the pair (`a`, `A`). In fact, there are context-free languages for which we cannot build a deterministic pushdown automaton, although we can build a non-deterministic one.

We should also mention that the pushdown automata as discussed here are a simplification of the ones we find in automata theory. In automata theory, pushdown automata have so-called states, and the mapping is from (state, input symbol, stack symbol) triplets to (state, list of stack symbols) pairs. Seen in this way, they are like finite-state automata (discussed in Chapter 5), extended with a stack. Also, pushdown automata come in two different kinds: some accept a sentence by empty stack, others accept by ending up in a state that is marked as an accepting state. Perhaps surpris-

ingly, having states does not make the pushdown automaton concept more powerful and pushdown automata with states still only accept languages that can be described with a context-free grammar. In our discussion, the pushdown automaton only has one state, so we have left it out.

Pushdown automata as described above have several shortcomings that must be resolved if we want to convert them into parsing automata. Firstly, pushdown automata require us to put our grammar into Greibach Normal Form. While grammar transformations are no problem for the formal-linguist, we would like to avoid them as much as possible, and use the original grammar if we can. Now we could relax the Greibach Normal Form requirement a little by also allowing terminals as stack symbols, and adding

$$(a,a) \rightarrow$$

to the mapping for all terminals a. We could then use any grammar all of whose right-hand sides start with a terminal. We could also split the steps of the pushdown automaton into separate "match" and "predict" steps, as we did in the example of Section 6.1. The "match" steps then correspond to usage of the

$$(a,a) \rightarrow$$

mappings, and the "predict" step then corresponds to a

$$(,A) \rightarrow \cdots$$

mapping, that is, a non-terminal on the top of the stack is replaced by one of its right-hand sides, without consuming a symbol from the input. For the grammar of Figure 6.1, this results in the mapping shown in Figure 6.4, which is in fact just a rewrite of the grammar of Figure 6.1.

```
(, S)  ->  aB
(, S)  ->  bA
(, A)  ->  a
(, A)  ->  aS
(, A)  ->  bAA
(, B)  ->  b
(, B)  ->  bS
(, B)  ->  aBB
(a, a)  ->
(b, b)  ->
```

Fig. 6.4. Match and predict mappings of the PDA for the grammar of Figure 6.1

We will see later that, even using this approach, we may have to modify the grammar anyway, but in the meantime this looks very promising, so we adopt this strategy. This strategy also solves another problem: ε-rules do not need special treatment any more. To get Greibach Normal Form, we would have to eliminate them but this is not necessary any more, because they now just correspond to a

$(,A) \rightarrow$

mapping.

The second shortcoming is that the pushdown automaton does not keep a record of the rules (mappings) it uses. Therefore, we introduce an *analysis stack* into the automaton. For every prediction step, we push the non-terminal being replaced onto the analysis stack, suffixed with the number of the right-hand side taken (numbering the right-hand sides of a non-terminal from 1 to n). For every match, we push the matched terminal onto the analysis stack. Thus, the analysis stack corresponds exactly to the parts to the left of the dashed line in Figure 6.2, and the dashed line represents the separation between the analysis stack and the prediction stack. This results in an automaton that at any point in time has a configuration as depicted in Figure 6.5. Such a configuration, together with its current state, stacks, etc. is called an *instantaneous description*. In Figure 6.5, matching can be seen as pushing the vertical line to the right.

matched input	rest of input
analysis	prediction

Fig. 6.5. An instantaneous description

The third and most important shortcoming, however, is the non-determinism. Formally, it may be satisfactory that the automaton accepts a sentence if and only if there is a sequence of choices that leads to an empty stack at the end of the sentence, but for our purpose it is not, because it does not tell us how to obtain this sequence. We have to guide the automaton to the correct choices. Looking back to the example of Section 6.1, we had to make a choice at several points in the derivation, and we did so based on some ad hoc considerations that were specific for the grammar at hand: sometimes we looked at the next symbol in the sentence, and there were also some points where we had to look further ahead, to make sure that there were no more `a`s coming. In the example, the choices were easy, because all the right-hand sides start with a terminal symbol. In general, however, finding the correct choice is much more difficult. The right-hand sides could for example equally well have started with a non-terminal symbol that again has right-hand sides starting with a non-terminal, etc.

In Chapter 8 we will see that many grammars still allow us to decide which right-hand side to choose, given the next symbol in the sentence. In this chapter, however, we will focus on top-down parsing methods that work for a larger class of grammars. Rather than trying to pick a choice based on ad hoc considerations, we would like to guide the automaton through all the possibilities. In Chapter 3 we saw that there are in general two methods for solving problems in which there are several alternatives in well-determined points: depth-first search and breadth-first search. We shall now see how we can make the machinery operate for both search methods. Since the number of actions involved can be exponential in the size of the input, even a small example can get quite big. To make things even more interesting, we will use the inherently

ambiguous language of Figure 3.4, whose grammar is here repeated in Figure 6.6, and we will use **aabc** as test input.

```
S_S  ->  AB | DC
A    ->  a | aA
B    ->  bc | bBc
D    ->  ab | aDb
C    ->  c | cC
```

Fig. 6.6. A more complicated example grammar

6.3 Breadth-First Top-Down Parsing

The breadth-first solution to the top-down parsing problem is to maintain a list of all possible predictions. Each of these predictions is then processed as described in Section 6.2 above, that is, if there is a non-terminal on top, the prediction stack is replaced by several new prediction stacks, as many as there are choices for this non-terminal. In each of these new prediction stacks, the top non-terminal is replaced by the corresponding choice. This prediction step is repeated for all prediction stacks it applies to (including the new ones), until all prediction stacks have a terminal on top.

For each of the prediction stacks we match the terminal in front with the current input symbol, and strike out all prediction stacks that do not match. If there are no prediction stacks left, the sentence does not belong to the language. So instead of one prediction (stack, analysis stack) pair, our automaton now maintains a list of prediction (stack, analysis stack) pairs, one for each possible choice, as depicted in Figure 6.7.

matched input	rest of input
analysis1	prediction1
analysis2	prediction2
...	...

Fig. 6.7. An instantaneous description of our extended automaton

The method is suitable for on-line parsing, because it processes the input from left to right. Any parsing method that processes its input from left to right and results in a leftmost derivation is called an *LL* parsing method. The first *L* stands for Left to right, and the second *L* for Leftmost derivation.

Now we almost know how to write a parser along these lines, but there is one detail that we have not properly dealt with yet: termination. Does the input sentence belong to the language defined by the grammar when, ultimately, we have an empty prediction stack? Only when the input is exhausted! To avoid this extra check, and to

avoid problems about what to do when we arrive at the end of sentence but have not finished parsing yet, we introduce a special so-called *end marker* **#**. This end marker is appended both to the end of the sentence and to the end of the prediction, so when both copies match we know that the prediction has been matched by the input and the parsing has succeeded.

(*a*)

	aabc#
	S#

(*b*)

	aabc#
S_1	DC#
S_2	AB#

(*c*)

	aabc#
S_1D_1	abC#
S_1D_2	aDbC#
S_2A_1	aB#
S_2A_2	aAB#

(*d*)

a	abc#
S_1D_1a	bC#
S_1D_2a	DbC#
S_2A_1a	B#
S_2A_2a	AB#

(*e*)

a	abc#
S_1D_1a	bC#
$S_1D_2aD_1$	abbC#
$S_1D_2aD_2$	aDbbC#
$S_2A_1aB_1$	bc#
$S_2A_1aB_2$	bBc#
$S_2A_2aA_1$	aB#
$S_2A_2aA_2$	aAB#

(*f*)

aa	bc#
$S_1D_2aD_1$a	bbC#
$S_1D_2aD_2$a	DbbC#
$S_2A_2aA_1$a	B#
$S_2A_2aA_2$a	AB#

(*g*)

aa	bc#
$S_1D_2aD_1$a	bbC#
$S_1D_2aD_2aD_1$	abbbC#
$S_1D_2aD_2aD_2$	aDbbbC#
$S_2A_2aA_1aB_1$	bc#
$S_2A_2aA_1aB_2$	bBc#
$S_2A_2aA_2aA_1$	aB#
$S_2A_2aA_2aA_2$	aAB#

(*h*)

aab	c#
$S_1D_2aD_1$ab	bC#
$S_2A_2aA_1aB_1$b	c#
$S_2A_2aA_1aB_2$b	Bc#

(*i*)

aab	c#
$S_1D_2aD_1$ab	bC#
$S_2A_2aA_1aB_1$b	c#
$S_2A_2aA_1aB_2bB_1$	bcc#
$S_2A_2aA_1aB_2bB_1$	bBcc#

(*j*)

aabc	#
$S_2A_2aA_1aB_1$bc	#

Fig. 6.8. The breadth-first parsing of the sentence **aabc**

6.3.1 An Example

Figure 6.8 presents a complete breadth-first parsing of the sentence **aabc**. At first there is only one prediction stack: it contains the start symbol and the end marker; no symbols have been accepted yet (frame *a*). The step leading to (*b*) is a predict step; there are two possible right-hand sides, so we obtain two prediction stacks. The difference of the prediction stacks is also reflected in the analysis stacks, where the different suffixes of **S** represent the different right-hand sides predicted. Another predict step with multiple right-hand sides leads to (*c*). Now all prediction stacks have a terminal on top; all happen to match, resulting in (*d*). Next, we again have some predictions with a non-terminal in front, so another predict step leads us to (*e*). The next step is a match step, and fortunately, some matches fail; these are dropped as they can never lead to a successful parse. From (*f*) to (*g*) is again a predict step. Another match in which again some matches fail leads us to (*h*). A further prediction results in (*i*) and then a match brings us finally to (*j*), leading to a successful parse with the end markers matching.

The analysis is

$$\mathbf{S}_2\mathbf{A}_2\mathbf{a}\mathbf{A}_1\mathbf{a}\mathbf{B}_1\mathbf{b}\mathbf{c}\#$$

For now, we do not need the terminals in the analysis; discarding them gives

$$\mathbf{S}_2\mathbf{A}_2\mathbf{A}_1\mathbf{B}_1$$

This means that we get a leftmost derivation by first applying rule $\mathbf{S}_2$, then rule $\mathbf{A}_2$, etc., all the time replacing the leftmost non-terminal. Check:

S → AB → aAB → aaB → aabc

The breadth-first method described here was first presented by Greibach [7]. However, in that presentation, grammars are first transformed into Greibach Normal Form, and the steps taken are like the ones our initial pushdown automaton makes. The predict and match steps are combined.

6.3.2 A Counterexample: Left Recursion

The method discussed above clearly works for this grammar, and the question arises whether it works for all context-free grammars. One would think it does, because all possibilities are systematically tried, for all non-terminals, in any occurring prediction. Unfortunately, this reasoning has a serious flaw, which is demonstrated by the following example: let us see if the sentence **ab** belongs to the language defined by the simple grammar

S → Sb | a

Our automaton starts off in the following state:

	ab#
	S#

As we have a non-terminal at the beginning of the prediction, we use a predict step, resulting in:

	ab#
S_1	Sb#
S_2	a#

As one prediction again starts with a non-terminal, we predict again:

	ab#
S_1S_1	Sbb#
S_1S_2	ab#
S_2	a#

By now, it is clear what is happening: we seem to have ended up in an infinite process, leading us nowhere. The reason for this is that we keep trying the **S→Sb** rule without ever coming to a state where a match can be attempted. This problem can occur whenever there is a non-terminal that derives an infinite sequence of sentential forms, all starting with a non-terminal, so no matches can take place. As all these sentential forms in this infinite sequence start with a non-terminal, and the number of non-terminals is finite, there is at least one non-terminal A occurring more than once at the start of those sentential forms. So we have: $A \rightarrow \cdots \rightarrow A\alpha$. A non-terminal that derives a sentential form starting with itself is called *left-recursive*.

Left recursion comes in several kinds: we speak of *immediate left recursion* when there is a grammar rule $A \rightarrow A\alpha$, like in the rule **S→Sb**; we speak of *indirect left recursion* when the recursion goes through other rules, for example $A \rightarrow B\alpha$, $B \rightarrow A\beta$. Both these forms of left recursion can be concealed by ε-producing non-terminals; this causes *hidden left recursion* and *hidden indirect left recursion*, respectively. For example in the grammar

```
S → ABc
B → Cd
B → ABf
C → Se
A → ε
```

the non-terminals **S**, **B**, and **C** are all left-recursive. Grammars with left-recursive non-terminals are called left-recursive as well.

If a grammar has no ε-rules and no loops, we could still use our parsing scheme if we use one extra step: if a prediction stack has more symbols than the unmatched part of the input sentence, it can never derive the sentence (every non-terminal derives at least one symbol), so it can be dropped. However, this little trick has one big disadvantage: it requires us to know the length of the input sentence in advance, so the method no longer is suitable for on-line parsing. Fortunately, left recursion can be eliminated: given a left-recursive grammar, we can transform it into a grammar without left-recursive non-terminals that defines the same language. As left recursion poses a major problem for any top-down parsing method, we will now discuss this grammar transformation.

6.4 Eliminating Left Recursion

We will first discuss the elimination of immediate left recursion. We will assume that ε-rules and unit rules already have been eliminated (see Sections 4.2.3.1 and 4.2.3.2). Now, let A be a left-recursive rule, and

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

be all the rules for A. None of the α_i are equal to ε, or we would have a rule $A \rightarrow A$, a unit rule. None of the β_j are equal to ε either, or we would have an ε-rule. The sentential forms generated by A using only the $A \rightarrow A\alpha_k$ rules all have the form

$$A\alpha_{k_1}\alpha_{k_2}\cdots\alpha_{k_j}$$

and as soon as one of the $A \rightarrow \beta_i$ rules is used, the sentential form no longer has an A in front. It has the following form:

$$\beta_i\alpha_{k_1}\alpha_{k_2}\cdots\alpha_{k_j}$$

for some i, and some $k_1, \ldots, k_j$, where j could be 0. These same sentential forms are generated by the following set of rules:

$$\begin{aligned} A_{head} &\rightarrow \beta_1 \mid \cdots \mid \beta_m \\ A_{tail} &\rightarrow \alpha_1 \mid \cdots \mid \alpha_n \\ A_{tails} &\rightarrow A_{tail}\, A_{tails} \mid \varepsilon \\ A &\rightarrow A_{head}\, A_{tails} \end{aligned}$$

or, without re-introducing ε-rules,

$$\begin{aligned} A_{head} &\rightarrow \beta_1 \mid \cdots \mid \beta_m \\ A_{tail} &\rightarrow \alpha_1 \mid \cdots \mid \alpha_n \\ A_{tails} &\rightarrow A_{tail}\, A_{tails} \mid A_{tail} \\ A &\rightarrow A_{head}\, A_{tails} \mid A_{head} \end{aligned}$$

where A_{head}, A_{tail}, and A_{tails} are newly introduced non-terminals. None of the α_i is ε, so A_{tail} does not derive ε, so A_{tails} is not left-recursive. A could still be left-recursive, but it is not immediately left-recursive, because none of the β_j start with an A. They could, however, derive a sentential form starting with an A.

In general, eliminating indirect left recursion is more complicated. The idea is that first the non-terminals are numbered, say $A_1, A_2, \ldots, A_n$. Now, for a left-recursive non-terminal A there is a derivation

$$A \rightarrow B\alpha \rightarrow \cdots \rightarrow C\gamma \rightarrow A\delta$$

with all the time a non-terminal at the left of the sentential form, and repeatedly replacing this non-terminal using one of its right-hand sides. All these non-terminals have a number associated with them, say $i_1, i_2, \ldots, i_m$, and in the derivation we get the following sequence of numbers: $i_1, i_2, \ldots, i_m, i_1$. Now, if we did not have any rules $A_i \rightarrow A_j\alpha$ with $j \leq i$, this would be impossible, because $i_1 < i_2 < \cdots < i_m < i_1$ is impossible.

The idea now is to eliminate all rules of this form. We start with A_1. For A_1, the only rules to eliminate are the immediately left-recursive ones, and we already

have seen how to do just that. Next, it is A_2's turn. Each production rule of the form $A_2 \rightarrow A_1\alpha$ is replaced by the production rules

$$A_2 \rightarrow \alpha_1\alpha \mid \cdots \mid \alpha_m\alpha$$

where

$$A_1 \rightarrow \alpha_1 \mid \cdots \mid \alpha_m$$

are the A_1-rules. This cannot introduce new rules of the form $A_2 \rightarrow A_1\gamma$ because we have just eliminated A_1's left-recursive rules, and the α_i's are not equal to ε. Next, we eliminate the immediate left-recursive rules of A_2. This finishes the work we have to do for A_2. Likewise, we deal with A_3 through A_n, in this order, always first replacing rules $A_i \rightarrow A_1\gamma$, then rules $A_i \rightarrow A_2\delta$, etc. We have to obey this ordering because for example replacing a $A_i \rightarrow A_2\delta$ rule could introduce a $A_i \rightarrow A_3\gamma$ rule, but not a $A_i \rightarrow A_1\alpha$ rule.

6.5 Depth-First (Backtracking) Parsers

The breadth-first method presented in the previous section has the disadvantage that it uses a great deal of memory. The depth-first method also has a disadvantage: in its general form it is not suitable for on-line parsing. However, there are many applications where parsing does not have to be done on-line, and then the depth-first method is advantageous since it does not need much memory.

In the depth-first method, when we are faced with a number of possibilities, we choose one and leave the other possibilities for later. First, we fully examine the consequences of the choice we just made. If this choice turns out to be a failure (or even a success, but we want all solutions), we roll back our actions until the present point and continue with the other possibilities.

Let us see how this search technique applies to top-down parsing. Our depth-first parser follows the same steps as our breadth-first parser, until it encounters a choice: a non-terminal that has more than one right-hand side lies on top of the prediction stack. Now, instead of creating a new analysis stack/prediction stack pair, it chooses the first right-hand side. This is reflected on the analysis stack by the appearance of the non-terminal involved, with suffix 1, exactly as it was in our breadth-first parser. This time however, the analysis stack is not only used for remembering the parse, but also for backtracking.

The parser continues in this way, until a match fails, or the end markers match. If the prediction stack is empty, we have found a parse, which is represented by the analysis stack; if a match fails, the parser will backtrack. This backtracking consists of the following steps: first, any terminal symbols at the end of the analysis stack are popped from this stack, and pushed back on top of the prediction stack. Also, these symbols are removed from the matched input and added to the beginning of the rest of the input. This is the reversal of the "match" steps. So backtracking over a terminal is done by moving the vertical line backwards, as is demonstrated in Figure 6.9. Then there are two possibilities: if the analysis stack is empty, there are no

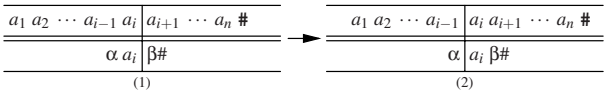


Fig. 6.9. Backtracking over a terminal

other possibilities to try, and the parsing stops; otherwise, there is a non-terminal on top of the analysis stack, and the top of the prediction stack corresponds to a right-hand side of this non-terminal. The choice of this right-hand side just resulted in a failed match. In this case we pop the non-terminal from the analysis stack and replace the right-hand side part in the prediction stack with this non-terminal. This is the reversal of a prediction step, as demonstrated in Figure 6.10. Next there are

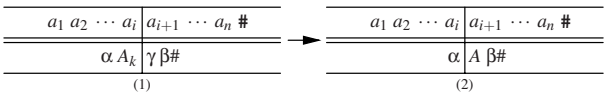


Fig. 6.10. Backtracking over the choice for the k-th rule for A, $A \rightarrow \gamma$

again two possibilities: if this was the last right-hand side of this non-terminal, we have already tried its right-hand sides and have to backtrack further; if not, we start parsing again, first using a predict step that replaces the non-terminal with its next right-hand side.

Now let us try to parse the sentence **`aabc`**, this time using the backtracking parser. Figure 6.11 presents the parsing process step by step; the backtracking steps are marked with a *B*. The example demonstrates another disadvantage of the backtracking method: it can make wrong choices and find out about this only much later.

As presented here, the parsing stops when a parsing is found. If we want to find all parsings, we should not stop when the end markers match. We can continue by backtracking just as if we had not found a successful parse, and record the analysis stack (which represents the parse) every time that the end markers match. Ultimately, we will end with an empty analysis part, indicating that we have exhausted all analysis possibilities, and the parsing stops.

6.6 Recursive Descent

In the previous sections, we have seen several automata at work, using a grammar to decide the parsing steps while processing the input sentence. Now this is just another way of stating that these automata use a grammar as a program. Looking at a grammar as a program for a parsing machine is not as far-fetched as it may seem. After all, a grammar is a prescription for deriving sentences of the language that the grammar describes, and what we are doing in top-down parsing is rederiving a sentence from the grammar. This only differs from the classic view of a grammar as a generating device in that we are now trying to rederive a particular sentence, not just

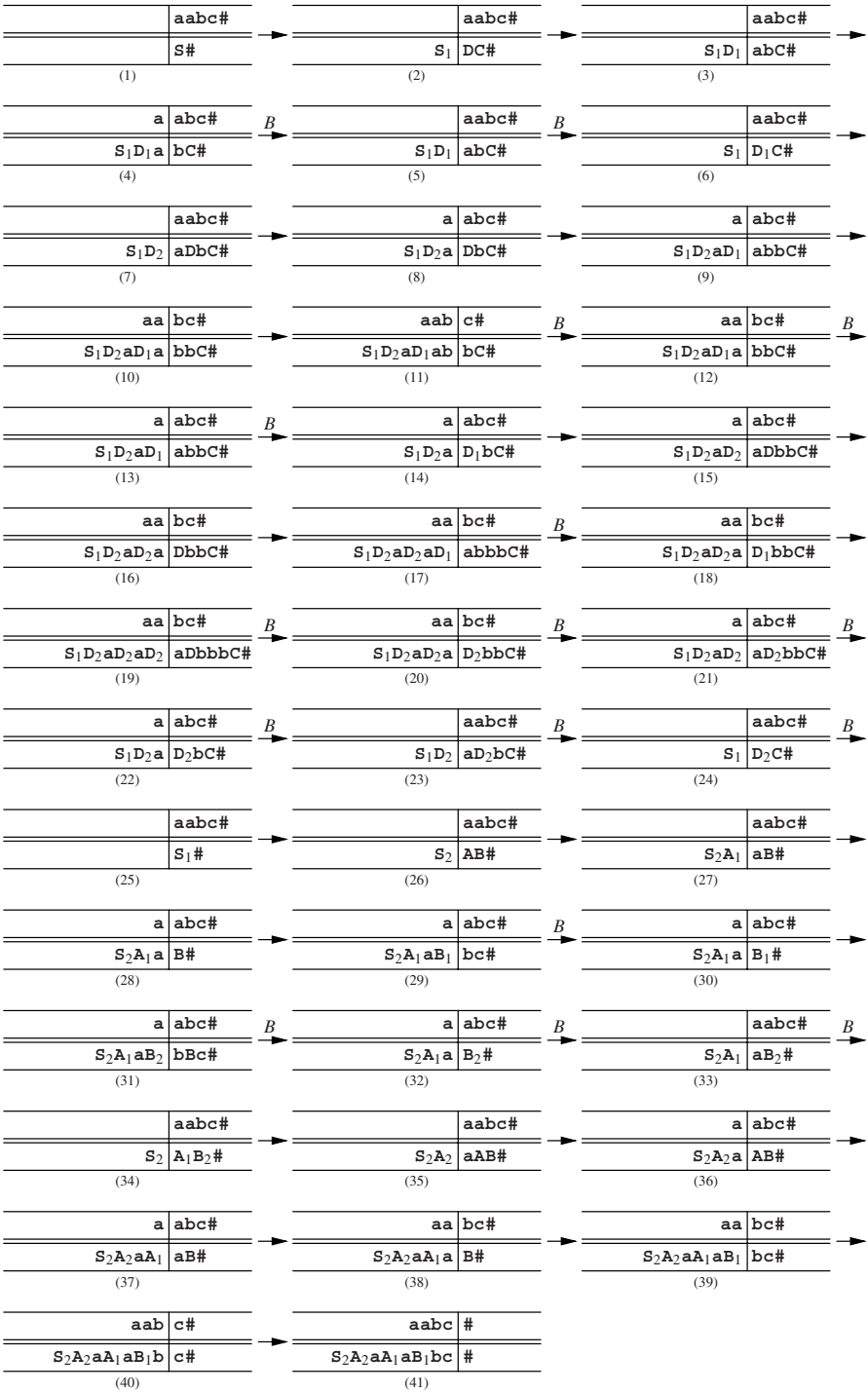

aabc# | S# (1)
aabc# | S1 DC# (2)
aabc# | S1D1 abC# (3)
a abc# | S1D1a bC# (4)
B
aabc# | S1D1 abC# (5)
B
aabc# | S1 D1C# (6)
aabc# | S1D2 aDbC# (7)
a abc# | S1D2a DbC# (8)
a abc# | S1D2aD1 abbC# (9)
aa bc# | S1D2aD1a bbC# (10)
aab c# | S1D2aD1ab bC# (11)
B
aa bc# | S1D2aD1a bbC# (12)
B
a abc# | S1D2aD1 abbC# (13)
B
a abc# | S1D2a D1bC# (14)
a abc# | S1D2aD2 aDbbC# (15)
aa bc# | S1D2aD2a DbbC# (16)
aa bc# | S1D2aD2aD1 abbbC# (17)
B
aa bc# | S1D2aD2a D1bbC# (18)
aa bc# | S1D2aD2aD2 aDbbbC# (19)
B
aa bc# | S1D2aD2a D2bbC# (20)
B
a abc# | S1D2aD2 aD2bbC# (21)
B
a abc# | S1D2a D2bC# (22)
B
aabc# | S1D2 aD2bC# (23)
B
aabc# | S1 D2C# (24)
B
aabc# | S1# (25)
aabc# | S2 AB# (26)
aabc# | S2A1 aB# (27)
a abc# | S2A1a B# (28)
a abc# | S2A1aB1 bc# (29)
B
a abc# | S2A1a B1# (30)
a abc# | S2A1aB2 bBc# (31)
B
a abc# | S2A1a B2# (32)
B
aabc# | S2A1 aB2# (33)
B
aabc# | S2 A1B2# (34)
aabc# | S2A2 aAB# (35)
a abc# | S2A2a AB# (36)
a abc# | S2A2aA1 aB# (37)
aa bc# | S2A2aA1a B# (38)
aa bc# | S2A2aA1aB1 bc# (39)
aab c# | S2A2aA1aB1b c# (40)
aabc # | S2A2aA1aB1bc # (41)

Fig. 6.11. Parsing the sentence **aabc**

any sentence. Seen in this way, grammars are programs, written in a programming language with a declarative style — that is, it defines the result but not the steps needed to obtain that result.

If we want to write a top-down parser for a given context-free grammar there are two options. The first is to write a program that emulates one of the automata described in the previous sections. This program can then be fed a grammar and an input sentence. This is a perfectly sound approach and is easy to program. The difficulty comes when the parser must perform some other actions as parts of the input are recognized. For example, a compiler must build a symbol table when it processes a declaration sequence. This, and efficiency considerations lead to the second option: writing a special purpose parser for the given grammar. Many of such special-purpose parsers have been written, and most of them use an implementation technique called *recursive descent*. We will assume that the reader has some programming experience, and knows about procedures and recursion. If not, this section can be skipped. It does not describe a new parsing method, but just an implementation technique that is often used in hand-written parsers and also in some machine-generated parsers.

6.6.1 A Naive Approach

As a first approach, we regard a grammar rule as a procedure for recognizing its left-hand side. A rule like

```
S  →  aB | bA
```

is regarded as a procedure to recognize an **S**. This procedure then states something like the following:

S succeeds if
  a succeeds and then **B** succeeds
or else
  b succeeds and then **A** succeeds

This does not differ much from the grammar rule, but it does not look like a piece of program either. Like a cookbook recipe, which usually does not tell us that we must peel the potatoes, let alone how to do that, the procedure is incomplete.

There are several bits of information that we must maintain when carrying out such a procedure. First, there is the notion of a "current position" in the rule. This current position indicates what must be tried next. When we implement rules as procedures, this current position is maintained automatically, by the program counter, which tells us where we are within a procedure. Next, there is the input sentence itself. When implementing a backtracking parser, we usually keep the input sentence in a global array, with one element for each symbol in the sentence. The array must be global, because it contains information that must be accessible equally easily from all procedures.

Then there is the notion of a current position in the input sentence. When the current position in the rule indicates a terminal symbol, and this symbol corresponds

to the symbol at the current position in the input sentence, both current positions will be advanced one position. The current position in the input sentence is also global information. We will therefore maintain this position in a global variable. Also, when starting a rule we must remember the current position in the input sentence, because we need it for the "or else" clauses. These must all be started at the same position in the input sentence. For example, starting with the rule for `S` of the grammar in Figure 6.1, suppose that the `a` matches the symbol at the current position of the input sentence. The current position is advanced and then `B` is tried. For `B`, we have a rule similar to that of `S`. Now suppose that `B` fails. We then have to try the next choice for `S`, and back up the position in the input sentence to what it was when we started the rule for `S`. This is backtracking, just as we have seen it earlier.

All this tells us how to deal with the procedure for one rule. However, usually we are dealing with a grammar that has more than one non-terminal, so there will be more than one rule. When we arrive at a non-terminal in a procedure for a rule, we have to call the procedure for that non-terminal, and, if it succeeds, return to the current invocation and continue there. We achieve this automatically by using the procedure-call mechanism of the implementation language.

Another detail that we have not covered yet is that we have to remember the grammar rules that we use. If we do not remember them, we will not know afterwards how the sentence was derived. Therefore we note them in a separate list, the "Parse list", striking them out when they fail. Each procedure must keep its own copy of the index in this list, again because we need it for the "or else " clauses: if a choice fails, all choices that have been made after the choice now failing must be discarded.

And the last detail to be filled in concerns the way the parser is started: by calling the procedure for the start symbol. When that procedure succeeds and the next symbol in the input is the end marker, the grammar rules left in the Parse list represent a leftmost derivation of the sentence.

Now let us see how a parser, as described above, works for an example. Let us consider again the grammar of Figure 6.6, and input sentence `abc`. We start with a call of `S` followed by a check for `#`, the input extended with `#`, and a Parse list consisting of just a node for `S`.

Active calls	Sentence	Parse list
1: •S#	•abc#	1: S

Our administration is divided into three parts. The "Active calls" part shows the active procedure calls, with a dot (•) indicating the current position within each call. The bottom rule in this part is the currently active procedure. The "Sentence" part indicates the sentence, including a position marker indicating the current position in the sentence. The "Parse list" will be used to remember the rules that we use (not only the currently active ones). The entries in this list are numbered, and each entry in the "Active calls" part also contains its index in the Parse list. As we shall see later, this is needed to back up after having taken a wrong choice.

Initially there is only one possibility: the current position in the active call indicates that we must invoke the procedure for `S`, so let us do so:

Active calls	Sentence	Parse list
1: S•#	•abc#	1: S
2: S → •DC | AB	•abc#	2: S → DC

Notice that we have advanced the position in the call of S. It now indicates where we have to continue when we are finished with S: the dot represents the return address. Now we try the first alternative for S. There is a choice here, so the current position in the input sentence is saved. We have not made this explicit in the pictures, because this position is already present in the "Sentence" part of the entry that invoked S.

Active calls	Sentence	Parse list
1: S•#	•abc#	1: S
2: S → D•C | AB	•abc#	2: S → DC
3: D → •ab | aDb	•abc#	3: D → ab

Now the first choice for D is tried. The a succeeds, and next the b also succeeds, so we get:

Active calls	Sentence	Parse list
1: S•#	•abc#	1: S
2: S → D•C | AB	•abc#	2: S → DC
3: D → ab• | aDb	ab•c#	3: D → ab

Now we are at the end of a choice for D. This means that the procedure for D succeeds and returns. We remove the entry from the list of active calls, after updating the current positions in the entry above. Next, it is C's turn to be called:

Active calls	Sentence	Parse list
1: S•#	•abc#	1: S
2: S → DC• | AB	ab•c#	2: S → DC
4: C → •c | cC	ab•c#	3: D → ab
		4: C → c

Now the c succeeds, so the C succeeds, and then the S also succeeds:

Active calls	Sentence	Parse list
1: S•#	abc•#	1: S
		2: S → DC
		3: D → ab
		4: C → c

Now the #s match, so we have found a parsing and the Parse list part represents a leftmost derivation of the sentence:

```
S → DC → abC → abc
```

This method is called *recursive descent*, "descent" because it operates top-down, and "recursive" because each non-terminal is implemented as a procedure that can directly or indirectly (through other procedures) invoke itself. It should be stressed that "recursive descent" is merely an implementation issue, albeit an important one.

It should also be stressed that the parser described above is a backtracking parser, independent of the implementation method used. Backtracking is a property of the parsing method, not of the implementation.

The backtracking method developed above is esthetically pleasing, because we in fact use the grammar itself as a program; we can actually transform the grammar rules into procedures mechanically, as we shall see below. There is only one problem: the recursive descent method, as described above, does not always work! We already know that it does not work for left-recursive grammars, but the problem is worse than that. For example, **aabc** and **abcc** are sentences that are not recognized, but should be. Parsing of the **aabc** sentence gets stuck after the first **a**, and parsing of the **abcc** sentence gets stuck after the first **c**. Yet, **aabc** can be derived as follows:

```
S → AB → aAB → aaB → aabc
```

and **abcc** can be derived with

```
S → DC → abC → abcC → abcc
```

So let us examine why our method fails. A little investigation shows that we never try the **A→aA** choice when parsing **aabc**, because the **A→a** choice succeeds. Likewise we never try **C→cD** when parsing **abcc**, because **C→c** succeeds. Such a problem arises whenever more than one right-hand side can succeed, and this is the case whenever a right-hand side can derive a prefix of a string derivable from another right-hand side of the same non-terminal. The method developed so far is too optimistic, in that it assumes that if a choice succeeds, it must be the right choice. It does not allow us to backtrack over such a choice, when it was the wrong one. This is a particularly serious problem if the grammar has ε-rules, because ε-rules always succeed. Another consequence of being unable to backtrack over a succeeding choice is that it does not allow us to get all parses when there is more than one (this is possible for ambiguous grammars).

Improvement is certainly needed here. Our criterion for determining whether a choice is the right one clearly is wrong. Looking back at the backtracking parser of the beginning of this section, we see that that parser does not have this problem, because it does not evaluate choices independently of their context. One can only decide that a choice is the right one if taking it results in a successful parse; even if the choice ultimately succeeds, we have to try the other choices as well if we want all parses. In the next section, we will develop a recursive-descent parser which solves all the problems mentioned above.

Meanwhile, the method above only works for grammars that are *prefix-free*. A non-terminal A is prefix-free if it does not produce two different strings x and y such that one is a prefix of the other. And a grammar is prefix-free if all its non-terminals are prefix-free. For the above algorithm this means that an attempt to recognize A in a given position in the input string can never succeed in more than one way, so if the first recognition is rejected later on, there is no need to go back and try another recognition because there will not be one.

6.6.2 Exhaustive Backtracking Recursive Descent

In the previous section we saw that we have to be careful not to accept a choice too early; it can only be accepted when it leads to a successful parse. Now this demand is difficult to express in a recursive-descent parser; how do we obtain a procedure that tells us whether a choice leads to a successful parse? In principle there are infinitely many of these, one for each sentential form (the prediction) that must derive the rest of the input, but we just cannot write them all. However, at any point during the parsing process we are dealing with only one such sentential form: the current prediction, so we could try to build a parsing procedure for this sentential form dynamically, during parsing. Some programming languages offer a useful facility for this purpose: procedure parameters. With procedure parameters, a procedure can accept another procedure (or even the same one) as a parameter and call it, or pass it on to another procedure.

Let us see how we can use procedure parameters to write a parsing procedure for a symbol X. This procedure for X is passed a procedure *tail* that parses the rest of the sentence, the part that follows the X. Such procedures are called *continuations*, since they embody the continuation of the work to be done. So a call $X(tail)$ will parse the entire input by first parsing X and then calling *tail* to parse the rest. This is the approach taken for all non-terminals, and, for the time being, for terminals as well.

The parsing procedure for a terminal symbol a is easy: it tries to match the current input symbol with a. If it succeeds, it advances the input position, and calls the *tail* parameter; then, when *tail* returns, it restores the input position and returns. If it fails it just returns. So the abstract code for a is

```
procedure a(tail):
begin if text[tp]='a' then begin tp:=tp+1; tail(); tp:=tp-1 end end;
```

where the input is in an array `text` and the input position in the variable `tp`.

The parsing procedure for a non-terminal A is more complicated. The simplest case is $A \rightarrow \varepsilon$, which is implemented as a call to *tail*. The next simple case is $A \rightarrow X$, where X is either a terminal or a non-terminal symbol. To deal with this case, we must remember that we assume that we have a parsing procedure for X, so the implementation of this case consists of a call to X, with the *tail* parameter.

The next case is $A \rightarrow XY$, with X and Y symbols. The procedure for X expects a procedure for "what comes after the X" as parameter. Here this parameter procedure is built using the Y and the *tail* procedures: we create a new procedure out of these two. This, by itself, is a simple procedure: it calls Y, with *tail* as a parameter. If we call this procedure Y_{tail}, we can implement A by calling X with Y_{tail} as parameter. So the abstract code for the rule $A \rightarrow XY$ is

```
procedure A(tail):
begin
    procedure Y_tail: begin Y(tail) end;
    X(Y_tail)
end;
```

And finally, if the right-hand side contains more than two symbols, this technique has to be repeated: for a rule $A \rightarrow X_1 X_2 \cdots X_n$ we create a procedure for $X_2 \cdots X_n$ and

tail using a procedure for $X_3 \cdots X_n$ and *tail*, and so on. So the abstract code for a rule $A \rightarrow X_1 X_2 \cdots X_n$ is

```
procedure A(tail):
begin
    procedure X_ntail: begin X_n(tail) end;
    ...
    procedure X_2...X_ntail: begin X_2(X_3...X_ntail) end;
    X_1(X_2...X_ntail)
end;
```

Here $X_2 \cdots X_{n\text{tail}}$, $X_3 \cdots X_{n\text{tail}}$, etc., are just names of new procedures. We see the prediction stack at the start of procedure X_n is represented by and encoded in the sequence of calls of X_1, $X_2 \cdots X_n$, and `tail`.

Finally, if we have a non-terminal with n alternatives, that is, we have $A \rightarrow \alpha_1 | \cdots | \alpha_n$, the parsing procedure for A has n consecutive code segments, one for each alternative, according to the above abstract code. They all call *tail* in their innermost procedures.

Applying this technique to all grammar rules results in a parser, except that we do not have a starting point yet. This is easily obtained: we just call the procedure for the start symbol, with the procedure for recognizing the end marker as a parameter.

This end-marker procedure is different from the others, because this is the procedure where we finally find out whether a parsing attempt succeeds. It tests whether we have reached the end of the input and if so, reports that we have found a parsing; it has no parameters, and so does not call any *tail*. Its abstract code is

```
procedure end_marker:
begin if at_end_of_input then report_parsing end;
```

The abstract code for the rule $A \rightarrow X_1 X_2 \cdots X_n$ declares the auxiliary procedures $X_2 \cdots X_{n\text{tail}}$ to $X_{n\text{tail}}$ as local procedures to that for A. This is necessary because *tail* must be accessible from $X_{n\text{tail}}$ and the only scope from which *tail* is accessible is inside the procedure for A. So to use this coding technique in practice we need a programming language that allows local procedures *and* allows them to be passed as parameters; unfortunately this rules out almost all present-day programming languages. The only reasonable possibilities are GNU C and the functional languages. GNU C is a widely available extension of C, and is used in the code below; parser writing in functional languages is treated briefly in Section 17.4.2. The technique can also be used in languages without local procedures, but then some trickery is required; see Problem 6.4.

Listings 6.13 and 6.14 present a fully backtracking recursive-descent parser for the grammar of Figure 6.6, written in GNU C. The program has a mechanism to remember the rules used (the procedures `pushrule()` and `poprule()` in Listing 6.14), so the rules can be printed for each successful parse. We see that, for example, the rule **`B->bBc`** corresponds to the code

```
static void c_t(void) { c(tail); }
static void Bc_t(void) { B(c_t); }
pushrule("B -> bBc"); b(Bc_t); poprule();
```

We have also used GNU C's facility to mix declarations and statements.

Figure 6.12 presents a sample session with this program. Note that no error mes-

```
> aabc
Derivation:
  S -> AB
  A -> aA
  A -> a
  B -> bc
> abcc
Derivation:
  S -> DC
  D -> ab
  C -> cC
  C -> c
> abc
Derivation:
  S -> DC
  D -> ab
  C -> c
Derivation:
  S -> AB
  A -> a
  B -> bc
> abca
```

Fig. 6.12. A session with the program of Listing 6.13

sage is given for the incorrect input **abca**; the parser just finds zero parsings.

We see that we can perform recursive descent by interpreting the grammar, as in Section 6.6.1, or by generating code and compiling it, as in Section 6.6.2. It is sometimes useful to make the distinction; the first can then be called *interpreted recursive descent* and the second *compiled recursive descent*.

6.6.3 Breadth-First Recursive Descent

Johnstone and Scott [36] present a different approach to exhaustive recursive descent, called *Generalized Recursive Descent Parsing* (GRDP). Like the naive approach of Section 6.6.1 it features a separate parsing procedure for each non-terminal. However, instead of returning as soon as a match is found, which was the pitfall causing the naive approach to fail, the GRDP procedure for a non-terminal A keeps track of all matches, and in the end returns the set of lengths of input segments that start at the current position and match A. The returned set can be empty, in which case no match was found.

The caller of such a procedure, which presumably is processing a right-hand-side in which A occurs, must be prepared for this and process each of the lengths in turn

```
static void a(void (*tail)(void)) /* recognize an 'a' and call tail */
{ if (text[tp] == 'a') { tp++; tail(); --tp; } }
static void b(void (*tail)(void)) /* recognize a 'b' and call tail */
{ if (text[tp] == 'b') { tp++; tail(); --tp; } }
static void c(void (*tail)(void)) /* recognize a 'c' and call tail */
{ if (text[tp] == 'c') { tp++; tail(); --tp; } }
static void A(void (*tail)(void)) /* recognize an 'A' and call tail */
{
        pushrule("A -> a"); a(tail); poprule();
        static void A_t(void) { A(tail); }
        pushrule("A -> aA"); a(A_t); poprule();
}
static void B(void (*tail)(void)) /* recognize a 'B' and call tail */
{
        static void c_t(void) { c(tail); }
        pushrule("B -> bc");  b(c_t); poprule();
        static void Bc_t(void) { B(c_t); }
        pushrule("B -> bBc"); b(Bc_t); poprule();
}
static void D(void (*tail)(void)) /* recognize a 'D' and call tail */
{
        static void b_t(void) { b(tail); }
        pushrule("D -> ab");  a(b_t); poprule();
        static void Db_t(void) { D(b_t); }
        pushrule("D -> aDb"); a(Db_t); poprule();
}
static void C(void (*tail)(void)) /* recognize a 'C' and call tail */
{
        pushrule("C -> c");  c(tail); poprule();
        static void C_t(void) { C(tail); }
        pushrule("C -> cC"); c(C_t); poprule();
}
static void S(void (*tail)(void)) /* recognize a 'S' and call tail */
{
        static void C_t(void) { C(tail); }
        pushrule("S -> DC"); D(C_t); poprule();
        static void B_t(void) { B(tail); }
        pushrule("S -> AB"); A(B_t); poprule();
}
static void endmark(void)      /* recognize end and report success */
{ if (text[tp] == '#') parsing_found(); }
static void parser(void) { tp = plp = 0; S(endmark); }
int main(void) { while ( getline()) parser(); return 0; }
```

Fig. 6.13. A fully backtracking recursive-descent parser for the grammar of Figure 6.6

```
#define MAXSIZE 100                 /* some large value */
/* text handling */
char text[MAXSIZE];                 /* input text */
int length;                         /* number of symbols in text */
int tp;                             /* index in text */
static int getline(void) {
        int ch;
        printf(">"); length = 0;
        while ((ch = getchar()), ch >= 0 && ch != '\n') {
                text[length++] = ch;
        }
        text[length] = '#'; return ch == '\n';
}
/* administration of rules used */
const char *parse_list[MAXSIZE];    /* store of rules used */
int plp;                            /* index in parse_list */
static void pushrule (const char *s) { parse_list[plp++] = s; }
static void poprule(void) { --plp; }
static void parsing_found(void) {
        int i;
        printf("Derivation:\n");
        for (i = 0; i < plp; i++) printf("    %s\n", parse_list[i]);
}
```

Fig. 6.14. Auxiliary code for the fully backtracking recursive-descent parser of Figure 6.13

when trying to match the rest of this right-hand-side. In the end, the caller of the procedure for the start symbol should check that the length of the input is a member of the returned set.

Given the grammar

$$\begin{aligned} \mathbf{S}_S &\rightarrow \mathbf{A\ a\ b} \\ \mathbf{A} &\rightarrow \mathbf{a\ A\ a} \mid \varepsilon \end{aligned}$$

and the input string **aaaaaaab**, the call in **S** to the routine for **A** will return the lengths 0, 2, 4, and 6, and only for length 6 will the routine for **S** be able to parse the remaining **ab**. The events inside the routine for **A** are more complex. After matching the first **a**, the routine calls itself, yielding the lengths 0, 2, 4, and 6. It tries these lengths and for each length it tries to match the final **a**; this succeeds for 0, 2, and 4, but not for 6. Together with the two matched **a**s this yields the lengths 2, 4, and 6. The alternative **A**→ε supplies the length 0, resulting in the length set { 0, 2, 4, 6 } as returned to **S**.

The fact that each procedure returns all possible matches prompted us to call this method breadth-first, although the method also has a depth-first aspect, in that each right-hand-side of a non-terminal is examined in-depth before the next right-hand-side is examined.

The method is suitable for all non-left-recursive CF grammars and can be optimized to perform competitively with common-place parser generators for LL(1)

grammars or non-left-recursive LR(1) grammars. The method is implemented in a freely available parser generator; see Johnstone and Scott [363].

6.7 Definite Clause Grammars

In Sections 6.6.1 and 6.6.2 we have seen how to create parsers that retain much of the original structure of the grammar. The programming language Prolog allows us to take this one step further. We will first give a very short introduction to Prolog and then explain how to create top-down parsers using it. For more information on Prolog, see, for example, *The Art of Prolog* by Leon Sterling and Ehud Shapiro (MIT Press).

6.7.1 Prolog

Prolog has its foundations in logic. The programmer declares some facts about objects and their relationships, and asks questions about these. The Prolog system uses a built-in search and backtracking mechanism to answer the questions. For example, we can tell the Prolog system about the fact that a table and a chair are pieces of furniture by writing

```
furniture(table).
furniture(chair).
```

We can then ask if a bread is a piece of furniture:

```
| ?- furniture(bread).
```

and the answer will be "no", but the answer to the question

```
| ?- furniture(table).
```

will be "yes". Such a Prolog form that can succeed or fail is called a *predicate* and when it is used as the start of a search it is called a *goal*.

We can also use variables, which can be either instantiated (have a value), or uninstantiated; such variables are called *logic variables*. Logic variables are identified by names starting with a capital letter or an underscore (_). We can use them for example as follows:

```
| ?- furniture(X).
```

This is asking for an *instantiation* of the logic variable `X`. The Prolog system will search for a possible instantiation and respond:

```
X = table
```

We can then either stop by typing a RETURN, or continue searching by typing a semicolon (and then a RETURN). In the latter case the Prolog system will search for another instantiation of `X`. The process of finding instantiations of logic variables that match the known facts is called *inference*.

Not every fact is as simple as the one in the example above. For example, a Prolog clause that tells us something about antique furniture is the following:

```
antique_furniture(Obj,Age) :- furniture(Obj), Age > 100.
```

Here we see a predicate which is the conjunction of two goals: an object **Obj** with age **Age** is an antique piece of furniture if it is a piece of furniture *and* its age is more than 100 years.

An important data structure in Prolog is the *list*, in which a arbitrary number of data items are concatenated. The empty list is denoted by **[]**; **[a]** is a list with head **a** and tail **[]**; **[a,b,c]** is a list with head **a** and tail **[b,c]**. Another useful data structure is the *compound value*, in which a fixed number of data items are combined in a named entity. An example is **dog('Fido',brown)**. Data items in data structures may be logic variables.

6.7.2 The DCG Format

Many Prolog systems allow us to specify grammars in a format that differs from the usual Prolog clauses. Since Prolog clauses are sometimes called *definite clauses*, a grammar in this format is called a *Definite Clause Grammar*, often abbreviated to *DCG*. The DCG form of the grammar of Figure 6.6 is shown in Figure 6.15. There

```
s_n  -->  d_n, c_n.
s_n  -->  a_n, b_n.
a_n  -->  [a].
a_n  -->  [a], a_n.
b_n  -->  [b], [c].
b_n  -->  [b], b_n, [c].
c_n  -->  [c].
c_n  -->  [c], c_n.
d_n  -->  [a], [b].
d_n  -->  [a], d_n, [b].
```

Fig. 6.15. The example grammar of Figure 6.6 in Definite Clause Grammar format

is a DCG predicate for each non-terminal and a DCG clause for each grammar rule. Since predicate names have to start with lower case letters in Prolog, we have translated non-terminal names like **S** by predicate names like **s_n**, for "**S**-non-terminal". The terminal symbols appear as lists of one element.

The Prolog system translates these DCG rules into Prolog clauses. The idea is to let each DCG rule for a non-terminal *A* correspond to a Prolog rule with two logic arguments of type list, traditionally called **Sentence** and **Remainder**, such that the rule

A`_n(Sentence, Remainder):- ...`

means that the character list **Sentence** is equal to whatever this rule for *A* produces concatenated with the character list **Remainder**.

More in particular, the DCG rule **d_n-->[a],[b].** corresponds to the Prolog clause

```
d_n(S,R) :- symbol(S,a,R1), symbol(R1,b,R).
```

where we have abbreviated `Sentence` to `S` and `Remainder` to `R`. The predicate `symbol()` is defined as

```
symbol([A|R],A,R).
```

This is a form of the Prolog predicate definition in which the condition lies in the matching of the arguments only: the predicate `symbol(S,a,R1)` succeeds when `S` can be split into two parts, `A` and `R`, such that `A` matches `a` and `R` matches R_1, in short when there is an R_1 such that $S=aR_1$. Likewise the predicate `symbol(R1,b,R)` tries to find an `R` such that $R_1=bR$. Together they enforce that $S=abR$, which is exactly what the DCG rule `d_n-->[a],[b].` means.

This technique can be extended to more than one intermediate logic variable, as, for example, in the translation of the second DCG rule for `d_n`:

```
d_n(S,R) :- symbol(S,a,R1), d_n(R1,R2), symbol(R2,b,R).
```

Here the Prolog system will have to find instantiations of two logic variables, R_1 and R_2 such that $S=aR_1$, $R_1=P$`(d_n)`R_2, and $R_2=bR$, where P`(d_n)` is any terminal production of `d_n`. When we combine these equations we obtain the semantics of `d_n(S,R)` as described above: $S=aP$`(d_n)`bR. (Most Prolog processors use a much less readable format internally.)

6.7.3 Getting Parse Tree Information

The DCG program of Figure 6.15 is a recognizer rather than a parser, but logic variables make it easy to collect parse tree information as well. To this end, we supply each non-terminal in the DCG program with a logic argument, the tree it has constructed. Nodes in the tree are conveniently represented as compound values, with the entire rule (between apostrophes) as the name and the children as the components. So a node for the rule $S \rightarrow AB$ with children `X` and `Y` is represented as `'S->AB'(X,Y)`. Tokens in right-hand sides do not produce parse trees, since they occur in the rule name already.

Since the parse trees of the children of a non-terminal A are delivered by the non-terminals of those children in the right-hand sides of the DCG rule for A, all we have to do to obtain the correct parse tree to be delivered by A is to create a compound value from the name of the rule and the parse trees of its children. The result is shown in Figure 6.16. It relies heavily on Prolog's ability to postpone the instantiation of values until the sources for that instantiation are available.

6.7.4 Running Definite Clause Grammar Programs

The DCG program of Figure 6.16 can be loaded into a Prolog interpreter, after which we can submit queries as described above. In composing these queries we should be aware that the root of the grammar, `S`, corresponds to a DCG name `s_n(T)` where `T` is the parse tree and to a Prolog predicate `s_n(T,S,R)` where `S` is the sentence

```
s_n('S→DC'(T1,T2))  -->  d_n(T1), c_n(T2).
s_n('S→AB'(T1,T2))  -->  a_n(T1), b_n(T2).
a_n('A→a')          -->  [a].
a_n('A→aA'(T1))     -->  [a], a_n(T1).
b_n('B→bc')         -->  [b], [c].
b_n('B→bBc'(T1))    -->  [b], b_n(T1), [c].
c_n('C→c')          -->  [c].
c_n('C→cC'(T1))     -->  [c], c_n(T1).
d_n('D→ab')         -->  [a], [b].
d_n('D→aDb'(T1))    -->  [a], d_n(T1), [b].
```

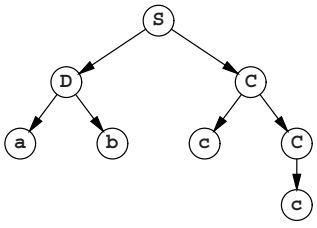
Fig. 6.16. The Definite Clause Grammar of Figure 6.16 with parse tree construction

and **R** is the remainder. The runs presented here were performed on C-Prolog version 1.5.

First we want the DCG program to generate some sentences, together with their parse trees. We do so by passing two uninstantiated variables, **S** and **T**, to **s_n**, and requesting the system to find three instantiations (user input has been underlined to differentiate it from computer output):

```
| ?- s_n(T,S,[]).
T = S->DC(D->ab,C->c)
S = [a,b,c] ;
T = S->DC(D->ab,C->cC(C->c))
S = [a,b,c,c] ;
T = S->DC(D->ab,C->cC(C->cC(C->c)))
S = [a,b,c,c,c] .
yes
```

We see that the system will only generate sentences **S** starting with an **a** followed by a **b**, and then followed by an ever increasing number of **c**s. The Prolog system uses a depth-first search mechanism, which is not suitable for sentence generation (see Section 2.4, where we use a breadth-first method). The values of **T** show the corresponding parse trees; each has the rule **S→DC** as the top, and the components describe its two children. It is important to note that **S→DC** is used here as a name, so **S→DC(D→ab,C→cC(C→c))** should be read as **'S->DC'('D->ab','C->cC'('C->c'))**, which corresponds to the parse tree



Next we ask the system to recognize some sentences; we start with our example from Section 6.6.1, **abc**:

```
| ?- s_n(T,[a,b,c],[]).
T = S->DC(D->ab,C->c) ;
T = S->AB(A->a,B->bc) ;
no
```

The full backtracking parser correctly parses the string but also supplies a second parsing, missed by the naive backtracking parser. The third answer, **no**, indicates that no further parsings are possible.

Finally we try the two input strings on which the naive parser of Section 6.6.1 failed, **aabc** and **abcc**:

```
| ?- s_n(T,[a,a,b,c],[]).
T = S->AB(A->aA(A->a),B->bc) ;
no
| ?- s_n([a,b,c,c],[]).
T = S->DC(D->ab,C->cC(C->c)) ;
no
```

Both input strings are handled correctly. They indeed involve applications of both **A→a** and **A→aA** in the first and of both **C→c** and **C→cC** in the second example.

These runs demonstrates that we can use Definite Clause Grammars quite well for recognizing sentences, and to a lesser extent also for generating sentences.

Cohen and Hickey [26] discuss this and other applications of Prolog in parsers in more detail.

6.8 Cancellation Parsing

We have pointed out repeatedly that top-down parsing cannot handle left recursion. This problem is shared by other tasks one would like to perform recursively, for example graph searching, where cycles in the graph can cause infinite loops. The standard solution there is to keep a set *B* of nodes that one is already visiting and back off when a node in *B* is about to be visited again. The set *B* (for "Busy") represents the subproblems already under examination, and starting to examine a subproblem (with the same parameters) while already examining it would lead to an infinite loop.

6.8.1 Cancellation Sets

The idea of "busy" sets can be used to make DCG parsing of left-recursive grammars possible (Nederhof [105]). Each Prolog predicate for a non-terminal *A* is given a third logic argument, **CancellationSet**, in addition to the original **Sentence** and **Remainder**. The *cancellation set* contains the names of the non-terminals that are already being investigated left-recursively, and plays the role of the "busy" set in graph handling. Using these sets results in *cancellation parsing*.

The first thing the rules for a non-terminal *A* do is to test whether *A* is in the cancellation set and back off if it is. This prevents looping on left recursion effectively, but as a result the rules for *A* will no longer recognize any terminal production of *A*

that involves left recursion. More precisely, they recognize only subtrees produced by A in which A figures nowhere on the "left spine" except at the top. Such subtrees must exist, or A produces nothing. So special measures must be taken to recognize parse trees in which A does occur more than once on the left spine. (The *left spine* of a tree is the sequence of nodes and arcs visited by starting at the top and following the leftmost branch at each node. Left spines are discussed in more detail in Section 10.1.1.1.)

The solution to this problem is simple but ingenious: as soon as a subtree T with A at the top is recognized, it is wrapped up to look like a newly defined terminal symbol, $\bar{A}$, and this terminal is pushed back into the input stream. In effect we scoop up a terminal production of A from the input, reduced it to a node A, and leave it in the input represented by the $\bar{A}$. Now the rest of the parser has to be adapted to the new situation. How this is done is explained in the next section.

6.8.2 The Transformation Scheme

We assume that the Prolog DCG system has the possibility to intersperse the DCG rules with "normal" Prolog text by putting it between curly braces, and that the underlying Prolog system has a definition for a predicate **`member(E,L)`** which succeeds when **`E`** is a member of the list **`L`**. The transformation scheme for a non-terminal A to a set of DCG cancellation rules then consists of three patterns, one for rules of the form $A \rightarrow B\alpha$, one for rules of the form $A \rightarrow t\alpha$, and one specific pattern to handle the $\bar{A}$:

```
A(C) --> {not member(A,C)}, B(A|C),
                    transformation of α, untoken(bar(A)), A(C).
A(C) --> {not member(A,C)}, [t],
                    transformation of α, untoken(bar(A)), A(C).
A(C) --> [bar(A)].
```

For the moment we assume that α is not empty; the complications with $A \rightarrow \varepsilon$ are treated in the next section.

This transformation scheme packs quite a few subtleties. The logic argument **`C`** is the cancellation set and the goal **`not member(A,C)`** implements the test whether A is already in the cancellation set. The transformation of the right-hand side follows the CF-DCG conversion shown in Section 6.7.2, except that all calls of non-terminals get a cancellation set as an argument. If the right-hand side starts with a non-terminal, that non-terminal gets a cancellation set equal to the old one extended with A; all other non-terminals get empty cancellation sets.

The DCG form **`untoken(bar(A))`** pushes back a copy of $\bar{A}$ into the input stream. It works as follows: the Prolog predicate **`untoken`** is defined as

```
untoken(T,S,[T|S]).
```

and the DCG processor will develop the DCG application **`untoken(a)`** into the Prolog goal **`untoken(a,Sentence,Remainder)`**. As a result, a call of **`untoken(a,Sentence,Remainder)`** will set **`Remainder`** to **`[a|Sentence]`**, thus prepending **`a`** to the rest of the input.

At this point in the pattern for A `(C)` we have recognized a $B\alpha$ or a $t\alpha$, reduced it and pushed it back as $\bar{A}$; so in terms of input we have not made any progress and still have an A to parse, since that is what the caller of A `(C)` expects. This parsing process must be able to absorb the $\bar{A}$ from the input and incorporate it in the parse tree. There are two candidates for this $\bar{A}$: a left-recursive rule for A, and the caller of A `(C)`. In the first case a new A node will be constructed on the left spine; in the second the left spine of A nodes ends here. The Prolog system must consider both possibilities. This can be achieved by introducing a predicate A_x which is defined both as all left-recursive rules of A and as $\bar{A}$. Also, to allow the new left-recursive rules to be activated, A_x must be called *without* the indication that A is already being investigated. So the transformation scheme we have now is

```
A_l(C) --> {not member(A,C)}, B(A|C),
                  transformation of α, untoken(bar(A)), A_x(C).
A_n(C) --> {not member(A,C)}, B(A|C),
                  transformation of α, untoken(bar(A)), A_x(C).
A_n(C) --> {not member(A,C)}, [t],
                  transformation of α, untoken(bar(A)), A_x(C).
A_x(C) --> A_l(C).
A_x(C) --> [bar(A)].
```

where A_l stands for all left-recursive rules for A and A_n for all its non-left-recursive ones.

As long as there is an $\bar{A}$ prepended to the input the only rules that can make progress are those that can absorb the $\bar{A}$. One candidate is A_x `(C)-->`A_l `(C)` and the other is the B in the first pattern. This B will usually be equal to A, but it need not be, if A is indirectly left-recursive; in that case a call of B will eventually lead to a call of an A_l. If the B is actually an A, its replacement in the transformation must be able to absorb an $\bar{A}$, and must still be able to parse a non-left-recursive instance of A. So we need yet another predicate here, A_b, defined by

```
A_b(C) --> A_n(C).
A_b(C) --> [bar(A)].
```

The fog that is beginning to surround us can be dispelled by a simple observation: we can add the non-left-recursive rules of A to A_x and the left-recursive ones to A_b, both without affecting the working of the parser, for the following reasons. The non-left-recursive rules of A can never absorb the $\bar{A}$, so adding them to A_x can at most cause failed calls; and calls to the left-recursive rules of A will be blocked by the left-recursion check preceding the A_b. So both A_x and A_b turn into a predicate A_t defined by

```
A_t(C) --> A(C).
A_t(C) --> [bar(A)].
```

In addition to simplifying the transformation scheme, this also removes the need to determine which rules are left-recursive.

This simplification leaves only occurrences of A and A_t in the transformation, where the As can occur only in non-first position in right-hand sides. In those positions they can be replaced by A_ts with impunity, since the only difference is that

A_t would accept a $\bar{A}$, but $\bar{A}$s do not occur spontaneously in the input. So in the end there are only A_ts left in the transformation patterns, which means that they can be renamed to A. This brings us to the transformation scheme at the beginning of the section.

Figure 6.17 shows the resulting cancellation parser for the grammar for simple arithmetic expressions of Figure 4.1. Notice that in

```
expr(C)  -->  {not member(expr,C)},
                  expr([expr|C]), ['+'], term([]),
                  {print('expr->expr+term'), nl},
                  untoken(bar(expr)), expr(C).
expr(C)  -->  {not member(expr,C)},
                  term([expr|C]),
                  {print('expr->term'), nl},
                  untoken(bar(expr)), expr(C).
term(C)  -->  {not member(term,C)},
                  term([term|C]), ['×'], factor([]),
                  {print('term->term×factor'), nl},
                  untoken(bar(term)), term(C).
term(C)  -->  {not member(term,C)},
                  factor([term|C]),
                  {print('term->factor'), nl},
                  untoken(bar(term)), term(C).
factor(C)  -->  {not member(factor,C)},
                  [i],
                  {print('factor->i'), nl},
                  untoken(bar(factor)), factor(C).
factor(C)  -->  {not member(factor,C)},
                  ['('], expr([]), [')'],
                  {print('factor->(expr)'), nl},
                  untoken(bar(factor)), factor(C).

expr(C)  -->  [bar(expr)].
term(C)  -->  [bar(term)].
factor(C)  -->  [bar(factor)].
```

Fig. 6.17. Cancellation parser in DCG notation for the grammar of Figure 4.1

`expr([expr|C]), ['+'], term([]),` the first `expr` is the name of a DCG predicate and the second is just a constant, to be added to the cancellation set.

Rather than building up the parse tree in a logic variable we produce it here using `print` statements; since these are placed at the end of the recognition, the tree is produced in bottom-up order. Running this DCG cancellation parser with the query `expr([],[i,'×',i,'+',i,'×',i],[])` yields the following reversed rightmost derivation:

```
factor->i
term->factor
factor->i
term->term×factor
expr->term
factor->i
term->factor
factor->i
term->term×factor
expr->expr+term
```

6.8.3 Cancellation Parsing with ε-Rules

Recognizing left-corner subtrees first has introduced a bottom-up component in our algorithm. We know that bottom-up techniques can have problems with ε-rules, and indeed, the naive transformation of a rule $A \rightarrow \varepsilon$,

```
A(C) --> {not member(A,C)}, untoken(bar(A)), A(C).
A(C) --> [bar(A)].
```

causes an infinite number of `bar(`A`)` tokens to be inserted in the input stream. As we wrote in Section 3.4.3.2, a bottom-up parser "will continue to find empty productions all over the place".

The technical reason for this failure is that the simplification applied above does not hold for ε-rules. An ε-rule is a non-left-recursive rule and although it cannot absorb the $\bar{A}$, it can succeed, and so provides a way for the $\bar{A}$ to stay in the input. This immediately suggests a solution: block the recognition of empty productions when there is already a barred token as first token of the rest of the input. To check this condition, we need a predicate `not_barred`, the Prolog form of which can be defined as

```
not_barred(S,S):- not(S = [bar(X)|T]).
```

It succeeds unless `S` can be decomposed in some barred head `X` and some tail `T`; note that the eventual values of `X` and `T` are immaterial.

The technique is applied in the following DCG cancellation parser for the grammar `S`→`Sa`|ε:

```
s_n(C)  -->  {not member(s_n,C)},
                 s_n([s_n|C]), [a],
                 untoken(bar(s_n)), s_n(C).
s_n(C)  -->  {not member(s_n,C)},
                 not_barred,
                 untoken(bar(s_n)), s_n(C).

s_n(C)  -->  [bar(s_n)].
```

6.9 Conclusion

General directional top-down parsing is quite powerful, but has a severe problem with left-recursive grammars. Although the recursive descent parsing method is elegant, the naive version is wrong and exhaustive backtracking recursive descent parsing requires some trickery.

Cancellation parsing allows for left-recursive grammars, and with some additional trickery will handle ε-rules as well.

General directional top-down parsers can easily be written in languages with good support for recursion, maintaining a close relationship between grammar and program code.

Problems

Problem 6.1: Prove that the grammar of Figure 6.1 produces *only* sentences with equal numbers of **a**s and **b**s, and that it produces *all* such sentences.

Problem 6.2: Non-deterministic PDAs like the ones in Figures 6.3 and 6.4 look somewhat like FSAs. Section 5.3.1 demonstrated how non-deterministic FSAs can be made deterministic, using the subset algorithm, but the fact that some transitions in PDAs stack more than one non-terminal prevents direct application of this technique. However, some stack only one non-terminal, and these could be made deterministic, resulting in sets of non-terminals on the prediction stack. Investigate this line of thought and compare it to the ideas of Sections 10.1.3 and 10.2.4.

Problem 6.3: *Research Project in Formal Languages*: *Hygiene in Pushdown Automata*: Find an algorithm that removes *useless transitions* from a given PDA. Several definitions of "useless transition" are possible; two reasonable ones are: a transition is useless if it *cannot* be used in any recognition of a correct input to the PDA; and: a transition is useless if it *need not* be used in any recognition of a correct input to the PDA. In both cases the removal of the transition does not affect the language recognized by the PDA.

Problem 6.4: Design a way to express the fully backtracking recursive descent parser from Figures 6.13 and 6.14 in an imperative or object-oriented language that does not allow local procedure declarations, for example ANSI C.

Problem 6.5: For the grammar of Figure 6.6, write a GRDP parser that keeps track of its actions so that it can produce parse trees, run it on the test input **aabc** and compare the results with Figure 6.11. Also run it on the test input **abc**, and confirm that it has two parses.

Problem 6.6: *Project*: The GRDP parser of Section 6.6.3 cannot handle left recursion, but it seems reasonable that that can be remedied as follows. Upon calling the routine for a non-terminal L we first suppress all left recursion; this gives us the set of lengths of segments (if present) that match L non-recursively. Then we call the routine for L again, now feeding it these lengths to use for the left-recursive call, so it can collect more; etc. In the end no more new matches are found, and the collected lengths can be returned. Turn this into a complete algorithm.

Problem 6.7: Run the DCG grammar of Figure 6.16 on a Prolog system and experiment with various correct and incorrect input.

Problem 6.8: (*a*) Modify the cancellation parser from Figure 6.17 to produce the parse tree in a logic variable rather than through `print` statements. (*b*) Make the grammar ambiguous by removing the precedence difference between `+` and `×`, using the rule `expr → expr [+ | ×] expr`, and experiment with it.

7

General Directional Bottom-Up Parsing

As explained in Section 3.2.2, directional bottom-up parsing is conceptually very simple. At all times we are in the possession of a sentential form that derives from the input text through a series of leftmost reductions. These leftmost reductions during parsing correspond to rightmost productions that produced the input text: the first leftmost *re*duction corresponds to the last rightmost *pro*duction, the second corresponds to the one but last, etc.

There is a cut somewhere in this sentential form which separates the already reduced part (on the left) from the yet unexamined part (on the right). See Figure 7.1. The part on the left is called the "stack" and the part on the right "rest of input". The

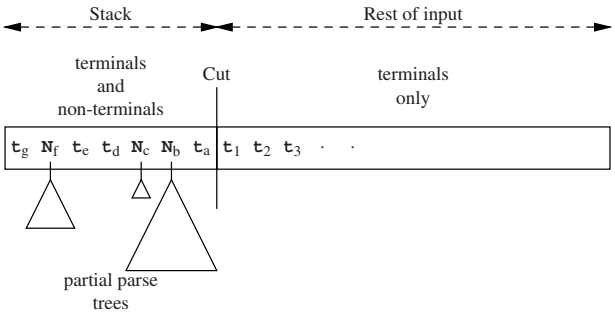


Fig. 7.1. The structure of a bottom-up parse

latter contains terminal symbols only, since it is an unprocessed part of the original sentence, while the stack contains a mixture of terminals and non-terminals, resulting from recognized right-hand sides. We can complete the picture by keeping the partial parse trees created by the reductions attached to their non-terminals. Now all the terminal symbols of the original input are still there; the terminals in the stack are one part of them, another part is semi-hidden in the partial parse trees and the rest is untouched in the rest of the input. No information is lost, but structure has been added. When the bottom-up parser has reached the situation where the rest of the

input is empty and the stack contains only the start symbol, we have achieved a parsing and the parse tree will be dangling from the start symbol. This view clearly exposes the idea that parsing is nothing but structuring the input.

The cut between stack and rest of input is often drawn as a gap, for clarity and since in actual implementations the two are often represented by quite different data structures in the parser. Note that the stack part corresponds to the open part of the sentential form when doing rightmost derivation, as discussed in Section 5.1.1.

Our non-deterministic bottom-up automaton can make only two moves: shift and reduce; see Figures 7.2 and 7.3. During a shift, a (terminal) symbol is shifted from the rest of input to the stack; $\mathtt{t}_1$ is shifted in Figure 7.2. During a reduce move, a

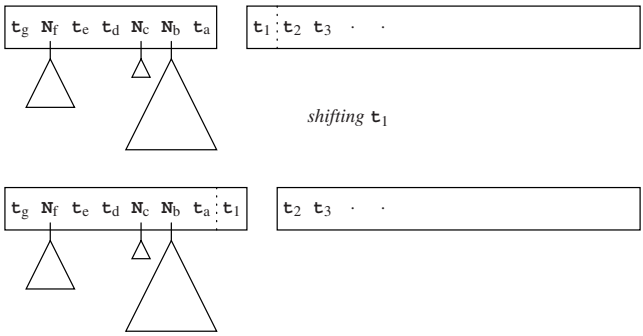


Fig. 7.2. A shift move in a bottom-up automaton

number of symbols from the right end of the stack, which form the right-hand side of a rule for a non-terminal, are replaced by that non-terminal and are attached to that non-terminal as the partial parse tree. $\mathtt{N}_c\mathtt{N}_b\mathtt{t}_a$ is reduced to $\mathtt{R}$ in Figure 7.3. We

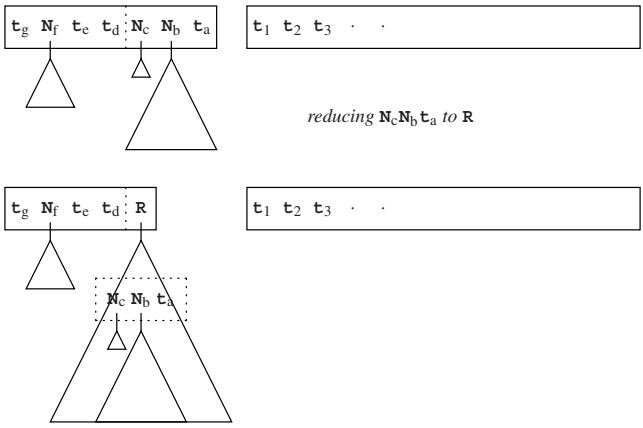


Fig. 7.3. A reduce move in a bottom-up automaton

see that the original $\mathtt{N}_c\mathtt{N}_b\mathtt{t}_a$ are still present inside the partial parse tree. There is, in principle, no harm in performing the instructions backwards, an *unshift* and an *unreduce*, although they would seem to move us away from our goal, which is to obtain a parse tree. We shall see that we need them to do backtracking.

At any point in time the machine can either shift (if there is an input symbol left) or not, or it can do one or more reductions, depending on how many right-hand sides can be recognized. If it cannot do either, it will have to resort to the backtrack moves, to find other possibilities. And if it cannot even do that, the parsing is finished, and the machine has found all (zero or more) parses.

We see that a parent node in the parse tree is identified after all its children have been identified: the parent `R` is not identified until each of its children $\mathtt{N}_c$, $\mathtt{N}_b$, and $\mathtt{t}_a$ have been recognized and put on the stack. This order of creating and visiting a tree is called "post-order".

7.1 Parsing by Searching

The only problem left is how to guide the automaton through all of the possibilities. This is easily recognized as a search problem, which can be handled by a depth-first or a breadth-first method. We shall now see how the machinery operates for both search methods. Since the effects are exponential in size, even the smallest example gets quite big and we shall use the unrealistic grammar of Figure 7.4. The test input is `aaaab`.

1. $\mathtt{S}_s \rightarrow \mathtt{a\ S\ b}$
2. $\mathtt{S} \rightarrow \mathtt{S\ a\ b}$
3. $\mathtt{S} \rightarrow \mathtt{a\ a\ a}$

Fig. 7.4. A simple grammar for demonstration purposes

7.1.1 Depth-First (Backtracking) Parsing

Refer to Figure 7.5, where the gap for a shift is shown as ⌈ and that for an unshift as ⌉. At first the gap is to the left of the entire input (frame *a*) and shifting is the only alternative; likewise with frame *b* and *c*. In frame *d* we have a choice, either to shift, or to reduce using rule 3. We shift, but remember the possible reduction(s); the rule numbers of these are shown as subscripts to the symbols in the stack. The same happens in frame *e*. In frame *f* we have reached a position in which the shift fails, the reduce fails (there are no right-hand sides `b`, `ab`, `aab`, `aaab`, or `aaaab`) and there are no stored alternatives on the `b`. So we start backtracking by unshifting (*g*). Here we find a stored alternative, "reduce by 3", which we apply (*h*), deleting the index for the stored alternative in the process. Now we can shift again (*i*). No more shifts are possible, but a reduce by rule 1 gives us a parsing (*j*), indicated by a ◁. After having

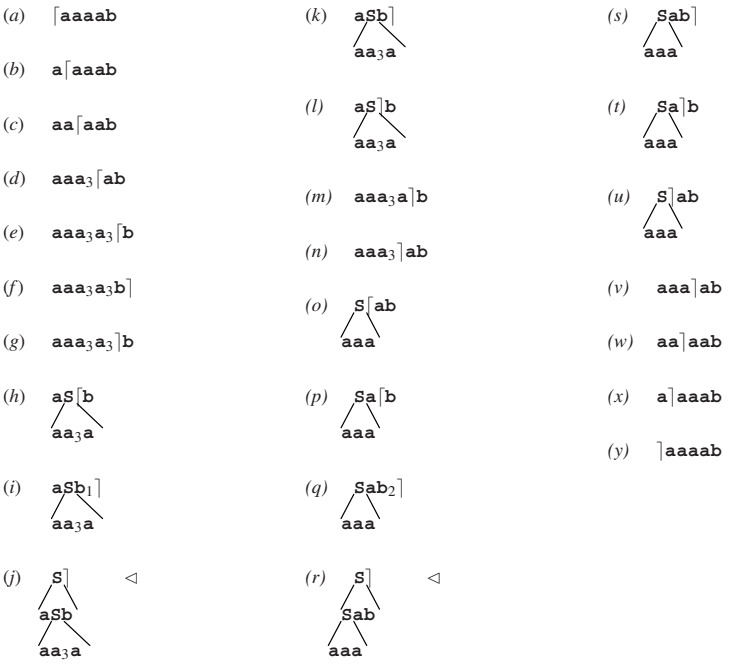


Fig. 7.5. Stages for the depth-first parsing of **aaaab**

enjoyed our success we unreduce (*k*); note that frame *k* only differs from frame *i* in that the stored alternative 1 has been consumed. Unshifting, unreducing and again unshifting brings us to frame *n* where we find a stored alternative, "reduce by 3". After reducing (*o*) we can shift again, twice (*p*, *q*). A "reduce by 2" produces the second parsing (*r*). The rest of the road is barren: unreduce, unshift, unshift, unreduce (*v*) and three unshifts bring the automaton to a halt, with the input reconstructed (*y*).

7.1.2 Breadth-First (On-Line) Parsing

Breadth-first bottom-up parsing is simpler than depth-first, at the expense of a far larger memory requirement. Since the input symbols will be brought in one by one (each causing a shift, possibly followed by some reduces), our representation of a partial parse will consist of the stack only, together with its attached partial parse trees. We shall never need to do an unshift or unreduce. Refer to Figure 7.6, where the gap is indicated by a (non-directional) $|$.

We start our solution set with only one empty stack (*a1*). Each parse step consists of two phases. In phase one the next input symbol is appended to the right of all stacks in the solution set; in phase two all stacks are examined and if they allow one or more reductions, one or more copies are made of it, to which the reductions are applied. This way we will never miss a solution. The first and second **a** are just appended (*b1*, *c1*), but the third allows a reduction (*d2*). The fourth causes one more

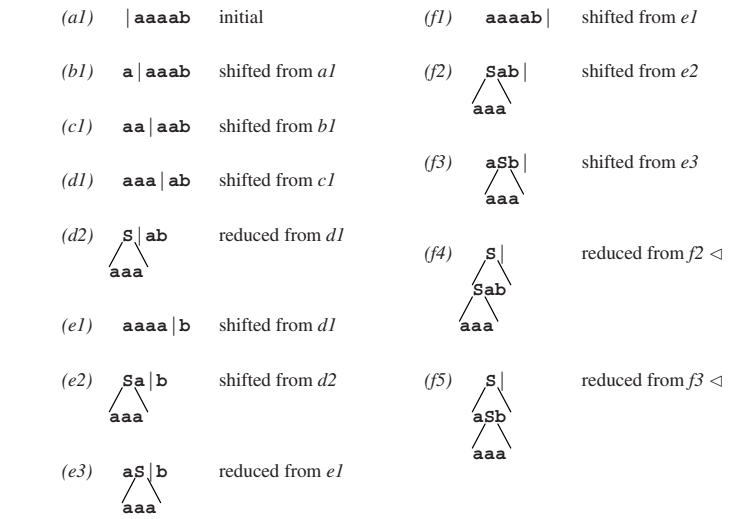


Fig. 7.6. Stages for the breadth-first parsing of **aaaab**

reduction (*e3*) and the fifth gives rise to two reductions, each of which produces a parsing (*f4* and *f5*).

7.1.3 A Combined Representation

The configurations of the depth-first parser can be combined into a single graph; see Figure 7.7(*a*) where numbers indicate the order in which the various shifts and reduces are performed. Shifts are represented by lines to the right and reduces by

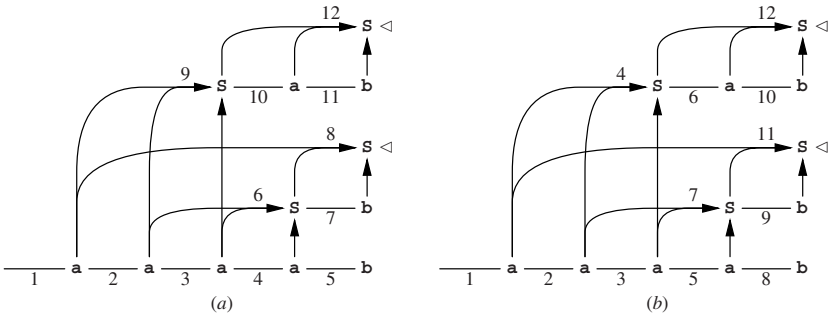


Fig. 7.7. The configurations of the parsers combined

upward arrows. Since a reduce often combines a number of symbols, the additional symbols are brought in by arrows that start upwards from the symbols and then turn right to reach the resulting non-terminal. These arrows constitute at the same time

the partial parse tree for that non-terminal. Start symbols in the rightmost column with partial parse trees that span the whole input head complete parse trees.

If we complete the stacks in the solution sets in our breadth-first parser by appending the rest of the input to them, we can also combine them into a graph, and, what is more, into the same graph; only the action order as indicated by the numbers is different, as shown in Figure 7.7(*b*). This is not surprising, since both represent the total set of possible shifts and reduces: depth-first and breadth-first are just two different ways to visit all nodes of this graph. Figure 7.7(*b*) was drawn in the same form as Figure 7.7(*a*). If we had drawn the parts of the picture in the order in which they are executed by the breadth-first search, many more lines would have crossed. The picture would have been equivalent to (*b*) but much more complicated.

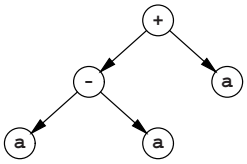
7.1.4 A Slightly More Realistic Example

The above algorithms are relatively easy to understand and implement; see, for example, Hext and Roberts [15] for Dömölki's method to find all possible reductions simultaneously. Although they require exponential time in general, they behave reasonably well on a number of grammars. Sometimes, however, they will burst out in a frenzy of senseless activity, even with an innocuous-looking grammar (especially with an innocuous-looking grammar!). The grammar of Figure 7.8 produces algebraic expressions in one variable, `a`, and two operators, `+` and `-`. `Q` is used for the

```
S_S  →  E
 E   →  E Q F
 E   →  F
 F   →  a
 Q   →  +
 Q   →  -
```

Fig. 7.8. A grammar for expressions in one variable

operators, since `O` (oh) looks too much like `0` (zero). This grammar is unambiguous and for `a-a+a` it has the correct production tree



which restricts the minus to the following `a` rather than to `a+a`. Figure 7.9 shows the graph searched while parsing `a-a+a`. It contains 109 shift lines and 265 reduce arrows and would fit on the page only thanks to the exceedingly fine print the phototypesetter is capable of. This is exponential explosion.

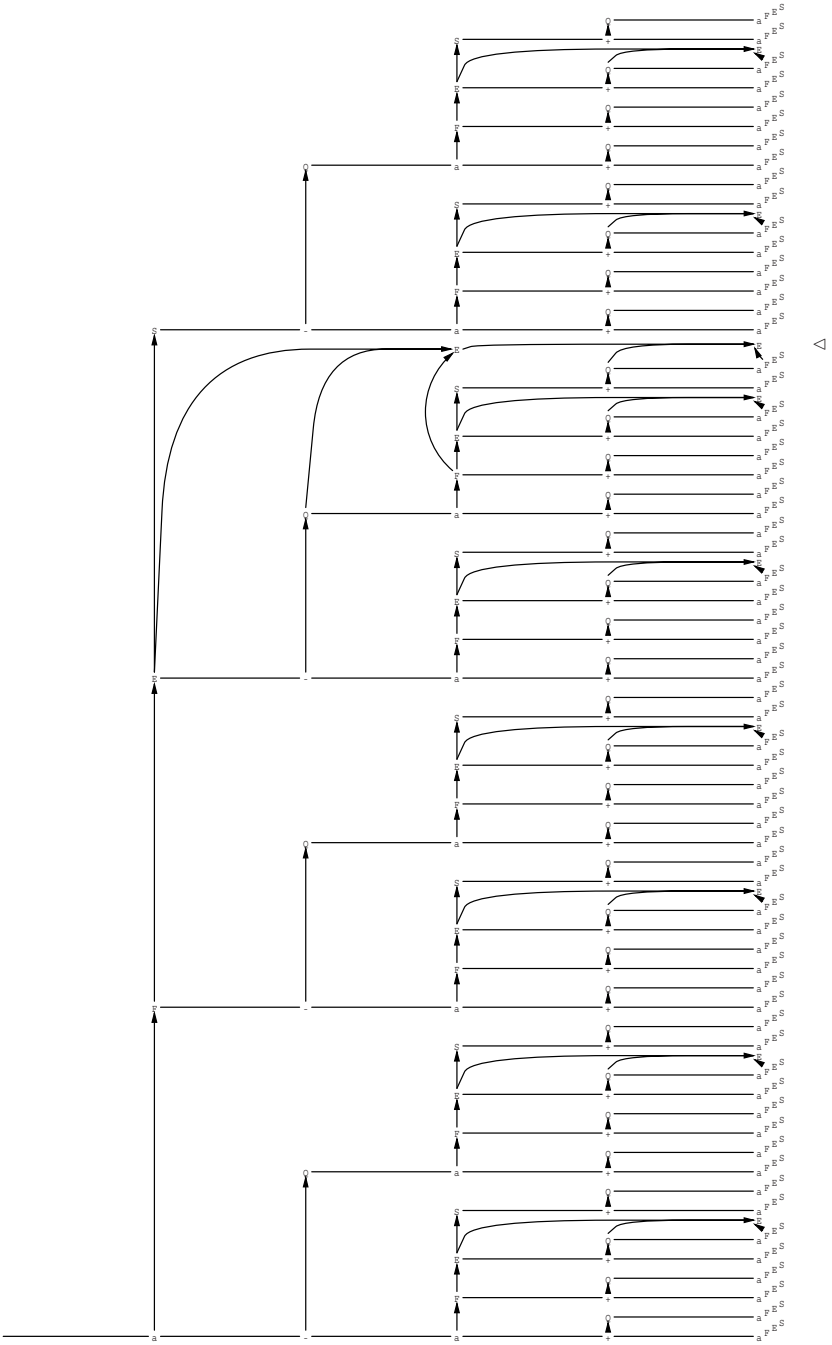


Fig. 7.9. The graph searched while parsing `a-a+a`

7.2 The Earley Parser

In spite of their occasionally vicious behavior, breadth-first bottom-up parsers are attractive since they work on-line, can handle left recursion without any problem and can generally be doctored to handle ε-rules and loops. So the question remains how to curb their needless activity. Many methods have been invented to restrict the search breadth to at most 1, at the expense of the generality of the grammars these methods can handle; see Chapter 9. A method that restricts the fan-out to reasonable proportions while still retaining full generality was developed by Earley [14].

7.2.1 The Basic Earley Parser

When we take a closer look at Figure 7.9, we see after some thought that many reductions are totally pointless. It is not meaningful to reduce the third **a** to **E** or **S** since these can only occur at the end if they represent the entire input; likewise the reduction of **a-a** to **S** is absurd, since **S** can only occur at the end. Earley noticed that what was wrong with these spurious reductions was that they were incompatible with a top-down parsing, that is: they could never derive from the start symbol. He then gave a method to restrict our reductions only to those that derive from the start symbol; the method is now known as *Earley parsing*. We shall see that the resulting parser takes at most n^3 units of time for input of length n rather than C^n for some constant C.

Earley's parser can also be described as a breadth-first top-down parser with bottom-up recognition, which is how it is explained by the author [14]. Since it can, however, handle left recursion directly but needs special measures to handle ε-rules, we prefer to treat it as a bottom-up method with a top-down component.

We shall again use the grammar from Figure 7.8 and parse the input **a-a+a**. Just as in the non-restricted algorithm from Section 7.1.1, we have at all times a set of partial solutions which is modified by each symbol we read. We shall write the sets between the input symbols as we go; we have to keep earlier sets, since they will still be used by the algorithm. Unlike the non-restricted algorithm, in which the sets contained stacks, the sets consist of what is technically known as *items*, or *Earley items* to be more precise. An "item" is a grammar rule with a gap in its right-hand side; the part of the right-hand side to the left of the gap (which may be empty) has already been recognized, the part to the right of the gap is predicted. The gap is traditionally shown as a fat *dot*: •. Examples of items are: **E→•EQF**, **E→E•QF**, **E→EQ•F**, **E→EQF•**, **F→a•**, etc. It is unfortunate that such a vague every-day term as "item" has become endowed with a very specific technical meaning, but the expression has taken hold, so it will have to do.

Items have quite different properties depending on exactly where the dot is, and the following types can be distinguished.

- An item with the dot at the end is called a *reduce item*, since the dot at the end means that the whole right-hand side has been recognized and can be reduced.
- An item with the dot at the beginning (just after the arrow) is known as a *predicted item*, since it results from a prediction, as we shall see below.

- An item with the dot in front of a terminal is called a *shift item*, since it allows a shift of the terminal.
- An item with the dot in front of a non-terminal does not have a standard name; we shall call it a *prediction item*, since it gives rise to predictions.
- An item with the dot not at the beginning is sometimes referred to as a *kernel item*, since at least part of it has been confirmed.

Although all items fall in at least one of the above classes, some fall in more than one: the types in this classification are not mutually exclusive. For example, the item **F→•a** is both a predicted and a shift item.

An *Earley item* is an item as defined above, with an indication of the position of the symbol at which the recognition of the recognized part started, its *origin position*. Notations vary, but we shall write @*n* after the item (read: "at *n*"). If the set at the end of position 7 contains the item **E→E•QF@3**, we have recognized an **E** in positions 3 through 7 and are looking forward to recognizing **QF**.

The sets of items contain exactly those items 1) of which the part before the dot has been recognized so far and 2) of which we are certain that we shall be able to use the result when they will happen to be recognized in full (but we cannot, of course, be certain that that will happen). For example, if a set contains the item **E→E•QF@3**, we can be sure that when we will have recognized the whole right-hand side **EQF**, we can go back to the set at the beginning of symbol number 3 and find there an item that was looking forward to recognizing an **E**, i.e., that had an **E** with a dot in front of it. Since that is true recursively, no recognition will be useless; of course, the recognized **E** is part of a right-hand side under construction and the full recognition of that right-hand side may eventually fail.

7.2.1.1 The Scanner, Completer and Predictor

The construction of an item set from the previous item set proceeds in three phases. The first two correspond to those of the non-restricted algorithm from Section 7.1.1, where they were called "shift" and "reduce"; here they are called "Scanner" and "Completer". The third is new and is related to the top-down component; it is called "Predictor".

The Scanner, Completer and Predictor operate on a number of interrelated sets of items for each token in the input. Refer to Figure 7.10, where the input symbol σ_p at position p is surrounded by several sets: $itemset_p$, which contains the items available just before σ_p; $completed_{p+1}$, the set of items that have become completed due to σ_p; $active_{p+1}$, which contains the non-completed items that passed σ_p; and $predicted_{p+1}$, the set of newly predicted items. The sets $active_{p+1}$ and $predicted_{p+1}$ together form $itemset_{p+1}$. Initially, $itemset_p$ is filled (as a result of processing σ_{p-1}) and the other sets are empty; $itemset_1$ is filled from the start symbol.

The Scanner looks at σ_p, goes through $itemset_p$ and makes copies of all items that contain $\bullet\sigma$; all other items are ignored. In the copied items, the part before the dot was already recognized and now σ is recognized; consequently, the Scanner changes $\bullet\sigma$ into $\sigma\bullet$. If the dot is now at the end, the Scanner has found a reduce item and stores it in the set $completed_{p+1}$; otherwise it stores it in the set $active_{p+1}$.

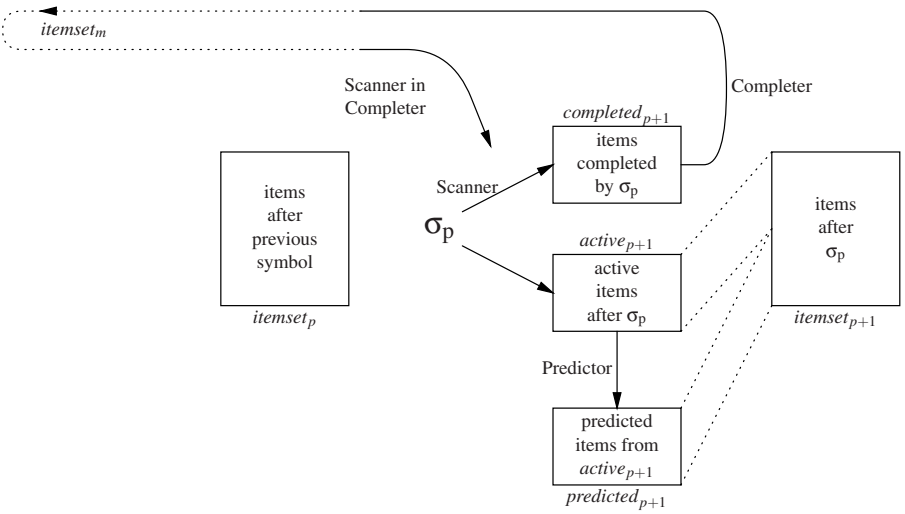


Fig. 7.10. The Earley item sets and their processes for one input symbol

Next the Completer inspects $completed_{p+1}$, which contains the items that have just been recognized completely and can now be reduced as follows. For each item of the form $R \rightarrow \cdots \bullet\ @m$ the Completer goes to $itemset_m$, and calls the Scanner in a special way as follows. The Scanner, which was used to working before on the terminal σ_p found in the input and $itemset_p$, is now directed to work on the non-terminal R recognized by the Completer and $itemset_m$. Just as for a terminal it copies all items in $itemset_m$ featuring a $\bullet R$, replaces the $\bullet R$ by $R\bullet$ and stores them in either $completed_{p+1}$ or $active_{p+1}$, as appropriate. This can add new recognized items to the set $completed_{p+1}$, which just means more work for the Completer. After a while, all completed items have been reduced, and the Predictor's turn has come.

The Predictor goes through the sets $active_{p+1}$, which was filled by the Scanner, and $predicted_{p+1}$, which is empty initially, and considers all items in which the dot is followed by a non-terminal. We expect to see these non-terminals in the input, and the Predictor predicts them as follows. For each such non-terminal N and for each rule for that non-terminal $N \rightarrow P \cdots$, the Predictor adds an item $N \rightarrow \bullet P \cdots\ @p+1$ to the set $predicted_{p+1}$. This may introduce new predicted non-terminals (for example P) in $predicted_{p+1}$ which cause more predicted items. After a while, this too will stop.

The sets $active_{p+1}$ and $predicted_{p+1}$ together form the new $itemset_{p+1}$. If the *completed* set after the last symbol in the input contains an item $S \rightarrow \cdots \bullet\ @1$, that is, an item spanning the entire input and reducing to the start symbol, we have found a parsing.

Now refer to Figure 7.11, which shows the item sets of the Earley parser working on **a-a+a** with the grammar from Figure 7.8. In this and following drawings, the

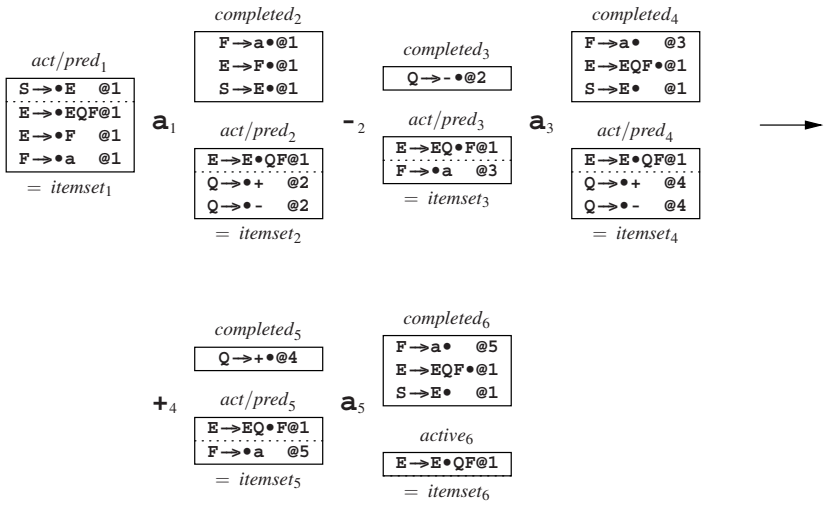


Fig. 7.11. Item sets of the Earley parser working on **a-a+a**

sets $active_p$, $predicted_p$, and $itemset_p$ have been combined into one set; the internal division between $active_p$ and $predicted_p$ is indicated in the drawings by a dotted line.

The initial active item set $active_1$ is {**S→•E@1**}, indicating that this is the only item that can derive directly from the start symbol. The Predictor first predicts **E→•EQF@1**, from this **E→•EQF@1** and **E→•F@1** (but the first one is in the set already) and from the last one **F→•a@1**. This gives $itemset_1$.

The Scanner working on $itemset_1$ and scanning for an **a**, only catches **F→•a@1**, turns it into **F→a•@1**, and stores it in $completed_2$. This means not only that we have recognized and reduced an **F**, but also that we have a buyer for it. The Completer goes to the set $itemset_1$ and copies all items that have **•F**. Result: one item, **E→•F@1**, which turns into **E→F•@1** and is again stored in $completed_2$. More work for the Completer, which will now copy items containing **•E**. Result: two items, **S→•E@1** which becomes **S→E•@1** and goes to the completed set, and **E→•EQF@1** which becomes **E→E•QF@1**, and which becomes the first and only member of $active_2$. The completion of **S** yields no new information.

The Predictor working on $active_2$ has an easy job: **•Q** causes two items for **Q**, both with **@2**, since that is where recognition will have started, if it occurs at all. Nothing spectacular happens until the Scanner processes the second **a**; from $itemset_3$ it extracts **F→•a@3** which gives **F→a•@3**, which is passed to the Completer (through $completed_4$). The latter sees the reduction of **a** to **F** starting at position 3, goes to $itemset_3$ to see who ordered an **F**, and finds **E→EQ•F@1**. Given the **F**, this turns into **E→EQF•@1**, which in its turn signals the reduction to **E** of the substring from 1 to 3 (again through $completed_4$). The Completer checks $itemset_1$ and finds two

clients there for the **E**: **S→•E@1** and **E→•EQF@1**; the first ends up as **S→E•@1** in $completed_4$, the second as **E→E•QF@1** in $active_4$.

After the last symbol has been processed by the Scanner, we still run the Completer to do the final reductions, but running the Predictor is useless, since there is nothing to predict any more. Note that the parsing started by calling the Predictor on the initial active set and that there is one Predictor/Scanner/Completer action for each symbol. Since $completed_6$ indeed contains an item **S→E•@1**, there is at least one parsing.

7.2.1.2 Constructing a Parse Tree

All this does not directly give us a parse tree. As is often the case in parser construction (see, for example, Section 4.1), we have set out to build a parser and have ended up building a recognizer. The intermediate sets, however, contain enough information about fragments and their relations to construct a parse tree easily. As with the CYK parser, a simple top-down Unger-type parser can serve for this purpose, since the Unger parser is very interested in the lengths of the various components of the parse tree and that is exactly what the sets in the Earley parser provide. In his 1970 article, Earley gives a method of constructing the parse tree(s) while parsing, by keeping with each item a pointer back to the item that caused it to be present. Tomita [162, p. 74-77] has, however, shown that this method will produce incorrect parse trees on certain ambiguous grammars.

The set $completed_6$ in Figure 7.11, which is the first we inspect after having finished the set construction, shows us that there is a parse possible with **S** for a root and extending over symbols 1 to 5; we designate the parse root as $\mathbf{S}_{1\text{-}5}$ in Figure 7.12. Given the completed item **S→E•@1** in $completed_6$ there must be a parse node $\mathbf{E}_{1\text{-}5}$, which is completed at 5. Since all items completed after 5 are contained in $completed_6$, we scan this set to find a completed **E** with origin position 1; we find **E→EQF•@1**. This gives us the parse tree in frame *a*, where the values at the question marks are still to be seen. Since items are recognized at their right ends, we start by finding a parse for the $\mathbf{F}_{?\text{-}5}$, to be found in $completed_6$. We find **F→a•@5**, giving us the parse tree in frame *b*. It suggests that we find a parse for $\mathbf{Q}_{?\text{-}4}$ completed after 4; in $completed_5$ we find **Q→+•@4**. Consequently $\mathbf{Q}_{?\text{-}4}$ is $\mathbf{Q}_{4\text{-}4}$ and the $\mathbf{E}_{1\text{-}?}$ in frame *b* must be $\mathbf{E}_{1\text{-}3}$. This makes us look in $completed_4$ for an **E→**$\cdots$**@1**, where we find **E→EQF•@1**. We now have a parse tree (*c*), and, using the same techniques, we easily complete it (*d*).

7.2.1.3 Space and Time Requirements

It is interesting to have a look at the space and time needed for the construction of the sets. First we compute the maximum size of the sets just after symbol number p. There is only a fixed number of different items, I, limited by the size of the grammar; for our grammar it is $I = 14$. However, each item can occur with any of the origin positions @1 to @$p+1$, of which there are $p+1$. So the number of items in the sets just after symbol number p is limited to $I \times (p+1)$. The exact computation of the

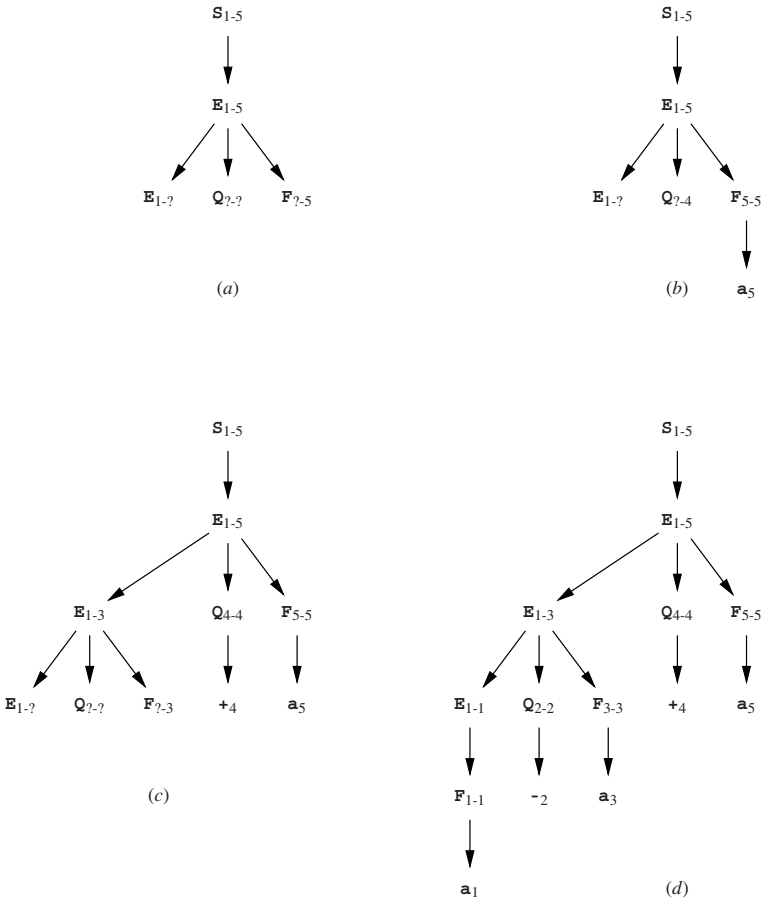


Fig. 7.12. Construction of the parse trees

maximum number of items in each of the sets is complicated by the fact that different rules apply to the first, last and middle items. Disregarding these complications, we find that the maximum number of items in all sets up to p is roughly $I \times p^2/2$. So, for an input of length n, the memory requirement is $O(n^2)$, as with the CYK algorithm. In actual practice, the amount of memory used is often far less than this theoretical maximum. In our case all sets together could conceivably contain about $14 \times 5^2/2 = 175$ items, with which the actual number of $4+3+3+1+2+3+3+1+2+3+1 = 26$ items compares very favorably.

Although a set at position p can contain a maximum of $O(p)$ items, it may require an amount of work proportional to p^2 to construct that set, since each item could, in principle, be inserted by the Completer once from each preceding position. Under the same simplifying assumptions as above, we find that the maximum number of actions needed to construct all sets up to p is roughly $I \times p^3/6$. So the total amount of work

involved in parsing a sentence of length n with the Earley algorithm is $O(n^3)$, as it is with the CYK algorithm. Again, in practice it is much better: on many grammars, including the one from Figure 7.8, it will work in linear time ($O(n)$) and on any unambiguous grammar it will work in $O(n^2)$. In our example, a maximum of about $14 \times 5^3/6 \simeq 300$ actions might be required, compared to the actual number of 28 (both items for **E** in $predicted_1$ were inserted twice).

It should be noted that once the computation of the sets is finished, only the *completed* sets are consulted. The *active* and *predicted* sets can be thrown away to make room for the parse tree(s).

The practical efficiency of this and the CYK algorithms is not really surprising, since in normal usage most arbitrary fragments of the input will not derive from any non-terminal. The sentence fragment "*letter into the upper leftmost*" does not represent any part of speech, nor does any fragment of size larger than one. The $O(n^2)$ and $O(n^3)$ bounds only materialize for grammars in which almost all non-terminals produce almost all substrings in almost all combinatorially possible ways, as for example in the grammar **S**→**SS**, **S**→**x**.

7.2.2 The Relation between the Earley and CYK Algorithms

The similarity in the time and space requirement between the Earley and the CYK algorithm suggest a deeper relation between the two and indeed there is: the Earley sets can be accommodated in a CYK-like grid, as shown in Figure 7.13. The horizontal axis of the CYK matrix represents the position where recognition started; its vertical level represents the length of what has been recognized. So an Earley item of the form $A \rightarrow \alpha \bullet \beta @ q$ in $itemset_p$ goes to column q, since that is where its recognition started, and to level $p - q$ since that is the length it has recognized. So the contents of an Earley set is distributed over a diagonal of the CYK matrix, slanting from north-west to south-east. Completed items are drawn in the top left corner of a box, active and predicted items in the bottom right corner. But since predicted items have not yet recognized anything they occur in the bottom layer only. When the reader turns Figure 7.13 clockwise over 45°, the Earley set can be recognized by stacking the boxes along the arrows at the bottom.

When we compare this picture to that produced by the CYK parser (Figure 7.14) we see correspondences and differences. Rather than having items, the boxes contain non-terminals only. All active and predicted items are absent. The left-hand sides of the completed items also occur in the CYK picture, but that parser features more recognized non-terminals; from the Earley picture we know that these will never play a role in any parse tree. The costs and the effects of the top-down restriction are clearly shown.

The correspondence between the Earley and CYK algorithms has been analysed by Graham and Harrison [19]. This has resulted in a combined algorithm described by Graham, Harrison and Ruzzo [23]. For a third, and very efficient representation of the CYK/Earley data structure see Kruseman Aretz [29].

There is another relationship between the Earley and CYK algorithms, which comes to light when the Earley data structure is expressed in tabular form (Section

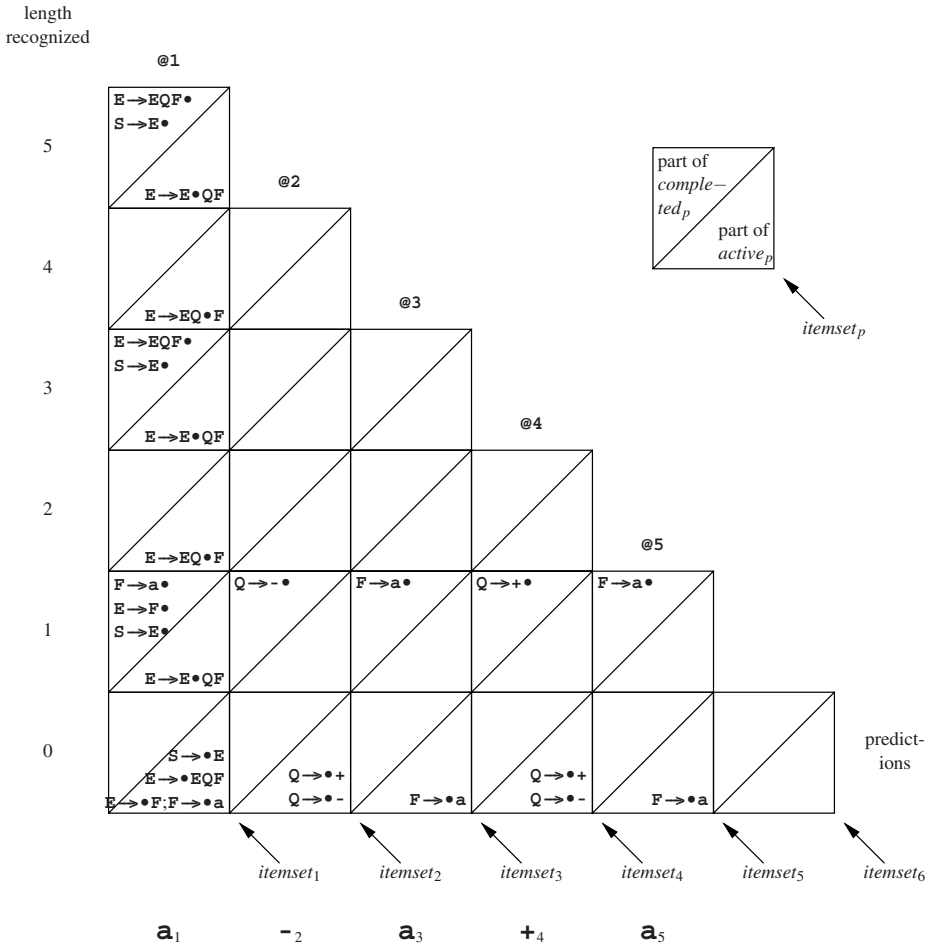


Fig. 7.13. The Earley sets represented in CYK fashion

4.3). The table is shown in Figure 7.15, where the vertical axis enumerates the items and the entries contain the lengths recognized for each item. We see that the entry for item **S→E•** at position 1 holds a 5 (among other lengths) indicating that the whole input can be parsed using rule **S→E** from position 1.

When we compare this table to the one in Figure 4.21, we see that the items take the positions of non-terminals, which suggests that they can be viewed as non-terminals in a grammar. And indeed they can. The grammar is shown in Figure 7.16; the items enclosed in { and } are names of new non-terminals, in spite of their looks.

We see that the grammar contains three kinds of rule. Those of the form $A \rightarrow \{A \rightarrow \bullet\alpha\} | \{A \rightarrow \bullet\beta\} | \cdots$ predict items for A and correspond to the Predictor. Those of the form $\{A \rightarrow \cdots \bullet X \cdots\} \rightarrow X\{A \rightarrow \cdots X \bullet \cdots\}$ correspond to the Scanner

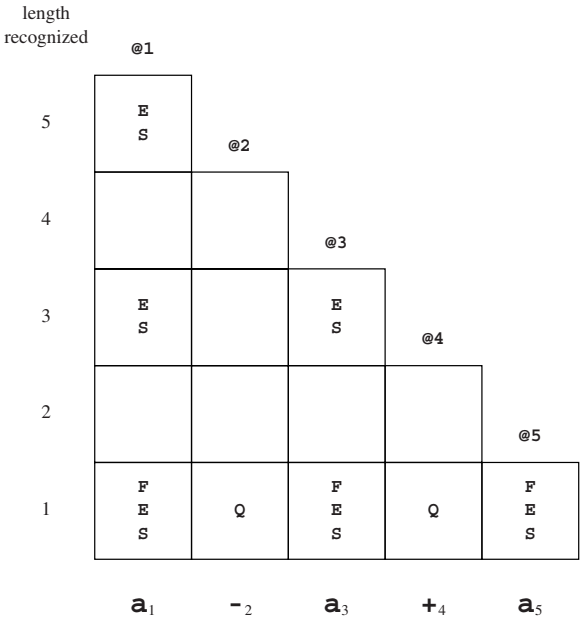


Fig. 7.14. CYK sets for the parsing of Figure 7.11

when X is a terminal and to the special Scanner inside the Completer when X is a non-terminal. And those of the form $\{A \rightarrow \cdots \bullet\} \rightarrow \varepsilon$ correspond to the Completer.

The most important point, however, is that no right-hand side in such an item grammar contains more than two non-terminals. As we have seen (page 116), the CYK parser has $O(n^3)$ (cubic) time requirements only when operated with a grammar with at most two non-terminals in any right-hand side. (Since such grammars give rise to binary trees as parse trees, they are said to be in *binary form*.) But the Earley parser has cubic time requirements for *any* grammar, and we now see why: the Completer/Scanner mechanism chops up the longer right-hand side into steps of 1, in a process similar to that in Section 4.2.3.4, thus creating a binary form grammar on the fly!

7.2.3 Handling ε-Rules

Like most parsers, the above parser cannot handle ε-rules without special measures. ε-rules show up first as an anomaly in the work of the Predictor. While predicting items of the form $A \rightarrow \bullet \cdots \ @p$ as a consequence of having a $\bullet A$ in an item in $active_p$ or $predicted_p$, it may happen to create an empty prediction $A \rightarrow \bullet \ @p$. This means that the non-terminal A has been completed just after symbol number p and this completed item should be added to the set $completed_p$, which up to now only contained items with origin position $p-1$ at most. So we find that there was more work for the Completer after all. But that is not the end of the story. If we now run the Completer again, it will draw the consequences of the newly completed item(s)

S→•E	0				
S→E•	1,3,5				
E→•EQF	0				
E→E•QF	1,3,5				
E→EQ•F	2,4				
E→EQF•	3,5				
E→•F	0				
E→F•	1				
F→•a	0		0		0
F→a•	1		1		1
Q→•+	0			0	
Q→+•				1	
Q→•-	0			0	
Q→-•		1			
	a 1	- 2	a 3	+ 4	a 5

Fig. 7.15. The Earley data structure in tabular form

```
{S→•E} → E {S→E•}          F → {F→•a}
{S→E•} → ε                {F→•a} → a {F→a•}
                          {F→a•} → ε
 E → {E→•EQF} | {E→•F}
{E→•EQF} → E {E→E•QF}      Q → {Q→•+} | {Q→•-}
{E→E•QF} → Q {E→EQ•F}     {Q→•+} → + {Q→+•}
{E→EQ•F} → F {E→EQF•}     {Q→+•} → ε
{E→EQF•} → ε              {Q→•-} → - {Q→-•}
{E→•F} → F {E→F•}         {Q→-•} → ε
{E→F•} → ε
```

Fig. 7.16. A grammar for Earley items

at origin position p. So it will consult $itemset_p$, which is, however, incomplete since items are still being added to its constituents, $active_p$ and $predicted_p$. If it finds items with occurrences of $\bullet A$ there, it will add copies with $A\bullet$ instead. Part of these may require new predictions to be made (if the dot lands in front of another non-terminal), and part of them may be completed items, which will have to go into $completed_p$ and which will mean more work for the Completer. The items in this set can have starting points lower than p, which bring in items from further back, to be added to $itemset_p$.

And of course these may or may not now be completed through this action or through empty completed items. Etc.

7.2.3.1 The Completer/Predictor Loop

The easiest way to handle this mare's nest is to stay calm and keep running the Predictor and Completer in turn until neither has anything more to add. Since the number of items is finite this will happen eventually, and in practice it happens sooner rather than later. (This is again a closure algorithm.)

The Completer and Predictor loop has to be viewed as a single operation called "X" by Graham, Harrison and Ruzzo [23]. Just like the Predictor it has to be applied to the initial state, to honor empty productions before the first symbol; just like the Completer it has to be applied to the final state, to honor empty productions after the last symbol.

Part of the effects are demonstrated by the grammar of Figure 7.17 which is based on a grammar similar to that of Figure 7.8. Rather than addition and subtraction, this

```
S_S  →  E
E    →  E Q F  |  F
F    →  a
Q    →  ×  |  /  |  ε
```

Fig. 7.17. A grammar with an ε-rule

one handles multiplication and division, with the possibility to omit the multiplication sign: **aa** means **a×a**.

The parsing is given in Figure 7.18. The items pointed at by a ▷ have been

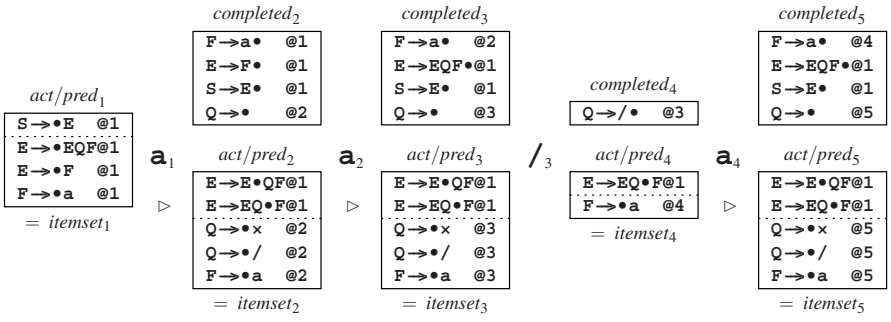


Fig. 7.18. Recognition of empty productions in an Earley parser

added by a second pass of the Completer/Predictor. The **Q→•@2**, inserted by the Predictor into *completed*$_2$ as a consequence of **E→E•QF@1** in *active*$_2$, is picked up by the second pass of the Completer, and is used to clone **E→E•QF@1** in *active*$_2$

into **E→EQ•F@1**. This in turn is found by the Predictor which predicts the item **F→•a@2** from it.

Note that we now do have to create the full *active/predicted* set after the last symbol, since its processing by the Completer/Predictor may insert an item of the form **S→···@1** in the last *completed* set, indicating a parsing.

7.2.3.2 Modifying the Predictor

Aycock and Horspool [38] show a way to avoid the Completer/Predictor loop. Before parsing we determine which non-terminals can produce the empty string, in other words, which non-terminals are nullable; we will see below in Section 7.2.3.3 how to do this. The processing of items during parsing is then arranged as follows:

- The items at a position p are put in a list, $list_p$. This list is initialized with the items produced by the Scanner moving items from $list_{p-1}$ over token σ_{p-1}.
- The items in the list are treated one by one by a Driver, in the order they appear in the list. If the item has the dot in front of a terminal, the Driver offers it to the Scanner; if the item has the dot in front of a non-terminal, the Driver offers it to the Predictor; otherwise the item has the dot at the end and is offered to the Completer. Any item resulting from this that must be inserted into the present list is added at the end, in order, unless it is already in the list, in which case it is discarded.
- The Predictor is modified as follows. When presented with an item $A \rightarrow \cdots \bullet B \cdots$ it predicts all items of the form $B \rightarrow \bullet \cdots$ as usual, but if B is nullable it also predicts the item $A \rightarrow \cdots B \bullet \cdots$.

It is clear that the loop has gone: each item gets treated exactly once. It is less obvious that this arrangement produces exactly the item sets that would have resulted from the Completer/Predictor loop.

When working on $list_p$, the Driver examines the items in turn and distributes them over the Scanner, the Predictor and the Completer. There is a fundamental difference between the Scanner and the Predictor on the one hand and the Completer on the other: the Scanner and the Predictor use only the item they are given (plus the next input token for the Scanner and the grammar rules for the Predictor), but the Completer combines it with other items from far-away and near-by places. The Completer takes one item of the form $A \rightarrow \cdots \bullet @q$ from $list_p$, goes to $list_q$, finds all items of the form $B \rightarrow \cdots \bullet A \cdots @r$ in that list and puts corresponding items $B \rightarrow \cdots A \bullet \cdots @r$ in $list_p$. Now that is fine as long as $q < p$, since then $list_q$ is already finished, and the scan will find all items. But when $q = p$, $list_q$, the list that is scanned, and $list_p$, the list under construction, are the same, and the Completer scan may miss items for the simple reason that they have not been appended yet.

The shortest grammar in which this problem occurs is

$$\begin{array}{lcl} \mathtt{S}_S & \rightarrow & \mathtt{A\ A\ x} \\ \mathtt{A} & \rightarrow & \varepsilon \end{array}$$

which produces only one string: **x**. Let us assume for the moment that we are still using the original Earley Predictor. The starting list $list_1$ is initialized with the item **S→•AAx@1**. The Driver examines it and passes it to the Predictor, which predicts the item **A→•@1** from it, which is appended to $list_1$. The Driver immediately picks it up, and offers it to the Completer, which scans $list_1$, combines the item with **S→•AAx@1** and produces **S→A•Ax@1**. The list $list_1$ now looks as follows:

```
S→•AAx@1
A→•@1
S→A•Ax@1
```

Next the Driver turns to **S→A•Ax@1**, and gives it to the Predictor. Again the predicted item **A→•@1** results, but since it is already in the list, it is not appended again. So the Driver does not find a new item, and stops.

We see that the resulting state does not accept the input token **x**. So it is wrong, but we are not surprised: the original algorithm would have gone on processing items until nothing changed any more. Soon it would have passed the item **A→•@1** to the Completer again, which would then have produced the item **S→AA•x@1**, and all would be well. In particular we see that one inference from the completed ε-item **A→•@1** is drawn, but later inferences are not, because each item is treated only once.

This is where the modified Predictor comes in: by predicting an item $A \to \cdots B\bullet\cdots@p$ from $A \to \cdots B\bullet\cdots@p$ when B is nullable, it provides the inference from $B \to \bullet$ (or any other item that causes B to produce ε) even when the item is out of sight because it has already been processed. Applied to the item **S→A•Ax@1** it produces *two* predictions: **A→•@1**, which is not appended, and **S→AA•x@1**, which is. This yields the correct $list_1$:

```
S→•AAx@1
A→•@1
S→A•Ax@1
S→AA•x@1
```

Aycock and Horspool [38] give a formal proof of the correctness of their algorithm. Figure 7.19 shows the lists for the same parsing as in Figure 7.18.

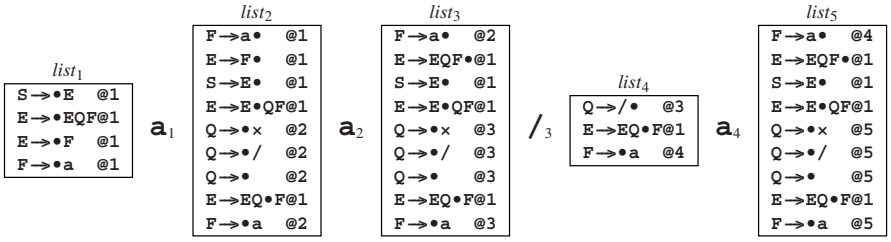


Fig. 7.19. Recognition of empty productions with a modified Predictor

Another way of avoiding the Completer/Predictor loop is to do ε-elimination on the grammar, as described in Section 4.2.3.1, but that would make the subsequent construction of the parse tree(s) much harder.

7.2.3.3 Determining Nullability

A simple closure algorithm allows us to find out which non-terminals in a grammar can produce the empty string (are nullable). First we scan the grammar and any time we find a rule of the form $A \rightarrow \varepsilon$ we mark A as nullable. Next we scan the grammar again and whenever we find a rule $P \rightarrow Q_1 \cdots Q_n$ where $Q_1 \cdots Q_n$ are all marked nullable, we mark P as nullable. Now we repeat the last step until no more non-terminals get marked. Then all nullable non-terminals have been marked.

7.2.4 Exploiting Look-Ahead

In the following paragraphs we shall describe a series of increasingly complicated (and more efficient) parsers of the Earley type. Somewhere along the line we will also meet a parser that is (almost) identical to the one described by Earley in his paper.

7.2.4.1 Prediction Look-Ahead

When we go back to Figure 7.11 and examine the actions of the Predictor, we see that it sometimes predicts items that it could know were useless if it could look ahead at the next symbol. When the next symbol is a `-`, it is kind of foolish to proudly predict `Q→•+@2`. The Predictor can of course easily be modified to check such simple cases, but it is possible to have a Predictor that will *never* predict anything obviously erroneous: all its predicted items will be either completed or active in the next set. Of course the predictions may fail on the symbol after that; after all, it is a Predictor, not an Oracle.

To see how we can obtain such a improved Predictor we need a different example, since after removing `Q→•+@2` and `Q→•-@4` from Figure 7.11 all predictions there come true, so nothing can be gained any more.

The artificial grammar of Figure 7.20 produces only the three sentences `p`, `q` and `pq`, and does so in a straightforward way. The root is `S'` rather than `S`, which is a convenient way to have a grammar with only one rule for the root. This is not necessary but it simplifies the following somewhat, and it is common practice.

The parsing of the sentence `q` is given in Figures 7.21(*a*) and (*b*). Since we are now using look-ahead, we have appended an end-of-input marker `#` to the input, as explained on page 94. Starting from the initial item, the Predictor predicts a list of 7 items (frame *a*). Looking at the next symbol, `q`, the Predictor could easily avoid the prediction `C→•p@1`, but several of the other predictions are also false, for example, `A→•C@1`. The Predictor could avoid the first since it sees that it cannot begin with `q`. If it knew that `C` cannot begin with a `q`, it could also avoid `A→•C@1`. (Note by the way that $itemset_2$ is empty, indicating that there is no way for the input to continue.)

```
S'_s  ->  S
S     ->  A | AB | B      FIRST(S) = {p, q}
                          FIRST(AB) = {p}
A     ->  C               FIRST(A) = {p}
B     ->  D               FIRST(B) = {q}
C     ->  p               FIRST(C) = {p}
D     ->  q               FIRST(D) = {q}
```

Fig. 7.20. A grammar for demonstrating prediction look-ahead and its FIRST sets

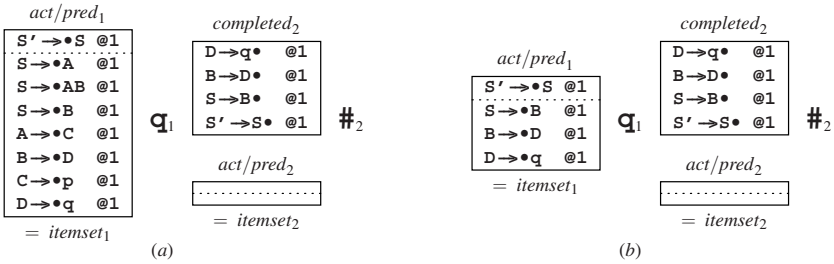


Fig. 7.21. Parsing the sentence **q** without look-ahead (*a*) and with look-ahead (*b*)

The required knowledge can be obtained by computing the FIRST sets of all non-terminals and their alternatives in the grammar. The FIRST set of a non-terminal *A* is the set of all tokens a terminal production of *A* can start with. Likewise, the FIRST set of an alternative α is the set of all tokens a terminal production of α can start with. These FIRST sets and a method of computing them are explained in Sections 8.2.1.1 and 8.2.2.1.

The FIRST sets of our grammar are shown in Figure 7.20. Since **S** has three alternatives, we need FIRST sets for each of them, to see which alternative(s) we must predict. FIRST(**A**) and FIRST(**B**) are already available as the FIRST sets of the non-terminals, but that of **AB** must be determined separately.

The use of the FIRST sets is very effective (frame *b*). The Predictor again starts from the initial item, but since it knows that **q** is not in FIRST(**A**) or FIRST(**AB**), it will avoid predicting **S→•A@1** and **S→•AB@1**, and just predict **S→•B@1**. Items like **A→•C@1** do not even have to be avoided, since their generation will never be contemplated in the first place. The item **S→•B@1** results in three predictions, all of them to the point.

As usual, ε-rules have a big impact. If we add a rule **C→**ε to our grammar (Figure 7.22), the entire picture changes. Starting from the initial item **S'→•S@1** (Figure 7.23), the Predictor will still not predict **S→•A@1** since FIRST(**A**) does not contain **q**, but it *will* predict **S→•AB@1** since FIRST(**AB**) does contain a **q**. Next **A→•C@1** is predicted, followed by **C→•@1**, but that is a completed item and goes into *completed*$_1$. When the Completer starts, it finds **C→•@1**, applies it to **A→•C@1** and produces **A→C•@1**, likewise completed. The latter is then applied to **S→•AB@1** to produce the active item **S→A•B@1**. This causes another run of the Predictor, to follow the new **•B**, but all those items have already been added.

S'_S	→	S		
S	→	A | AB | B		FIRST(S) = {ε, p, q}
				FIRST(AB) = {p, q}
A	→	C		FIRST(A) = {ε, p}
B	→	D		FIRST(B) = {q}
C	→	p | ε		FIRST(C) = {ε, p}
D	→	q		FIRST(D) = {q}

Fig. 7.22. A grammar with an ε-rule and its FIRST sets

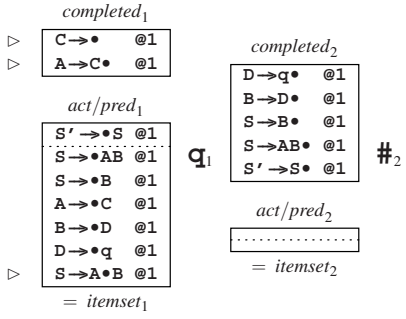


Fig. 7.23. Parsing the sentence q with the grammar of Figure 7.22

An interesting problem occurs when we try to parse the empty sentence, or actually the sentence #, since an end marker is appended. If we follow the above algorithm, we find that the look-ahead token # is not in any of the FIRST sets, for the simple reason that it is not part of the grammar, so no rule gets predicted, and the input is rejected. One way to solve the problem is to decide to predict an item only when the look-ahead does not contradict it, rather than when the look-ahead confirms it. A FIRST set containing ε does not contradict the look-ahead # (in fact it does not contradict any look-ahead), so the rules S→A, A→C, and C→ε get predicted. The resulting parsing is shown in Figure 7.24; we see that $completed_1$ con-

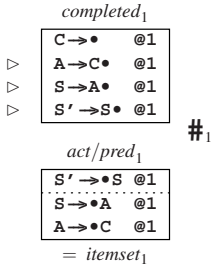


Fig. 7.24. Parsing the empty sentence with the grammar of Figure 7.22

tains **S→A•@1**, so the input is recognized, as expected. The next section shows a way to improve this algorithm and indeed predict only what the look-ahead confirms.

7.2.4.2 Reduction Look-Ahead

Once we have gone through the trouble of computing the FIRST sets, we can use them for a second type of look-ahead: *reduction look-ahead*. Prediction look-ahead reduces the number of predicted items, reduction look-ahead reduces the number of completed items. Referring back to Figure 7.11, which depicted the actions of an Earley parser without look-ahead, we see that it does two silly completions: **S→E•@1** in $completed_2$, and **S→E•@1** in $completed_4$. The redundancy of these completed items stems from the fact that they are only meaningful at the end of the input. Now this may seem a very special case, not worth testing for, but the phenomenon can be viewed in a more general way: if we introduce an explicit symbol for end-of-file (for example, **#**), we can say that the above items are redundant because they are followed by a symbol (**-** and **+**, respectively) which is not in the set of symbols that may follow the item on completion.

The idea is to keep, together with each item, a set of symbols which follow after that item, the reduction look-ahead set; if the item is a reduce item but the next symbol is not in this set, the item is not completed but discarded. The rules for constructing the look-ahead set for an item are straightforward, but unlike the prediction look-ahead it cannot be computed in advance; it must be constructed as we go. (A limited and less effective set could be computed statically, using the FOLLOW sets explained in Section 8.2.2.2.)

The initial item starts with a look-ahead set of **[#]** (look-ahead sets will be shown between square brackets at the end of items). When the dot advances in an item, its look-ahead set remains the same, since what happens inside an item does not affect what may come after it; only when a new item is created by the Predictor, a new look-ahead set must be composed. Suppose the parent item is

$$P \rightarrow A \bullet BCD[abc]\ @n$$

and predicted items for B must be created. We now ask ourselves what symbols may follow the occurrence of B in this item. It is easy to see that they are:

- any symbol C can start with,
- if C can produce the empty string, any symbol D can start with,
- if D can also produce the empty string, any of the symbols a, b and c.

Given the FIRST sets for all non-terminals, which can also tell us if a non-terminal can produce empty, the resulting new reduction look-ahead set is easily computed. It is also written as FIRST(CD $[abc]$), which is of course the set of first symbols of anything produced by $CDa|CDb|CDc$.

The Earley sets with reduction look-ahead for our example **a-a+a** are given in Figure 7.25. The computation of the sets follows the above rules. The look-ahead of the item **E→•EQF[#+-]@1** in $predicted_1$ results from its being inserted twice. Initially it is inserted by the Predictor from **S→•E[#]@1**, which contributes

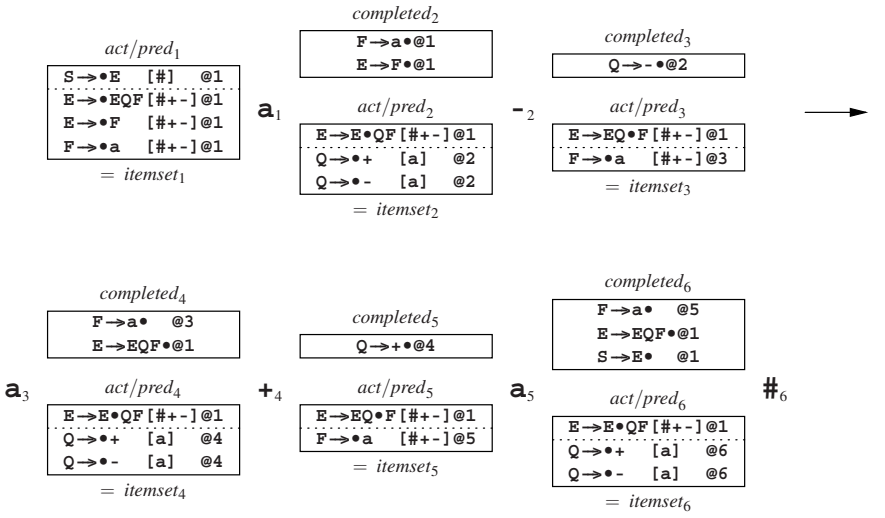


Fig. 7.25. Item sets with reduction look-ahead

the look-ahead `#`, and which results in the item `E→•EQF[#]@1`. When the Predictor processes this item, it predicts items for the `•E` in it, with a look-ahead of FIRST(`QF[#]`); this contributes `+-`. These items include `E→•EQF[+-]@1`, which together with the item from `S→•E[#]@1` results in the first item we see in $predicted_1$.

Note that the item `S→E•[#]@1` is not placed in $completed_2$, since the actual symbol ahead (`-`$_2$) is not in the item's look-ahead set; something similar occurs in $completed_4$, but not in $completed_6$.

Now that we have reduction look-ahead sets available in each item, we can use them to restrict our predictions to those confirmed by the look-ahead. Refer again to the grammar of Figure 7.22 and the parsing in Figure 7.24. The initial item is $\mathtt{S}'_s \rightarrow \bullet\mathtt{S[\#]}$, which gives rise to three potential items: `S→•A[#]`, `S→•AB[#]`, and `S→•B[#]`. Now we get FIRST(`A[#]`) = {`#`,`p`}, FIRST(`AB[#]`) = {`p`,`q`}, and FIRST(`B[#]`) = {`q`}. And since the look-ahead is `#`, only the first item survives. This improvement does not affect our example in Figure 7.24, but in general this use of the reduction look-ahead set in the prediction of items creates fewer items, and is thus more efficient.

7.2.4.3 Discussion

As with prediction look-ahead, the gain of reduction look-ahead in our example is meager, but that is mainly due to the unnatural simplicity of our example. The effectiveness of look-aheads in Earley parsers in the general case is not easily determined.

Bouckaert, Pirotte and Snelling [17], who have analysed variants of the Earley parsers for two different look-ahead regimes, show that prediction look-ahead reduces the number of items by 20 to 50% or even more on "practical" grammars.

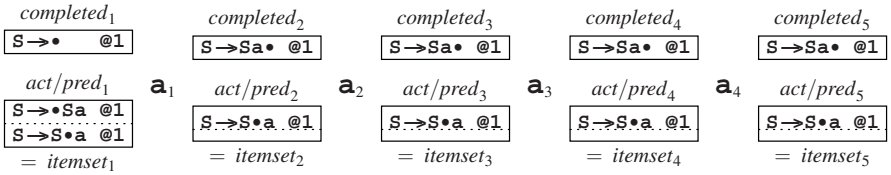
Earley recommends the reduction look-ahead, but does not take into account the effort required to compute and maintain the look-ahead sets. Bouckaert, Pirotte and Snelling definitely condemn the reduction look-ahead, on the grounds that it may easily double the number of items to be carried around, but they count, for example, **E→•F[+-]@1** as two items. All in all, since the gain from reduction look-ahead cannot be large and its implementation cost and overhead are probably considerable, it is likely to be counterproductive in practice.

The well-tuned Earley/CYK parser by Graham, Harrison and Ruzzo [23] features no look-ahead at all, claiming that more speed can be gained by efficient data structures and that carrying around look-ahead information would interfere with doing so.

McLean and Horspool [35] describe an optimized Earley parser, grouping the Earley items into subsets corresponding to LR states.

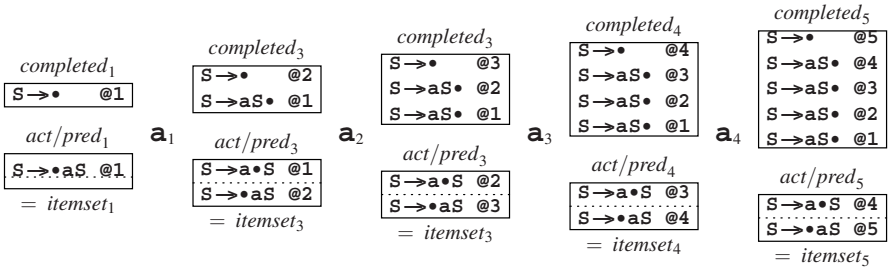
7.2.5 Left and Right Recursion

It is interesting to see how the Earley parser reacts to left-recursive and right-recursive grammars. As examples we will use the simple grammars **S→Sa** | ε (left-recursive) and **S→aS** | ε (right-recursive) on the input **aaaa...**. The Earley parser handles left recursion extremely well:



We see that the result is dull but very efficient: each next item set is constructed with a constant number of actions, so the parser takes linear time. (Note by the way that after the single prediction in $itemset_1$ no further predictions occur; so all items have origin position 1.)

The behavior on the right-recursive grammar is quite different:



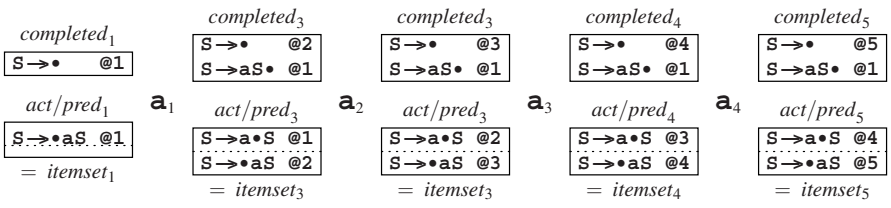
The number of completed items grows linearly because the parser recognizes n **S**s in position n: **a(a(a(a...)))**. This means that the parser has to perform $1+2+3+$

$4 + \cdots + n = n(n+1)/2$ actions; so the time requirements are quadratic. Although this is much better than the $O(n^3)$ from the general case, it is much worse than linear, and somehow it seems wasteful.

When we follow the algorithm through this example, we see that it collects and keeps a lot of information it never uses again. Let us look at position 4 where we have just shifted over $\mathbf{a}_3$, which caused the **S→•aS@3** in $itemset_3$ to be transformed into **S→a•S@3** and to be inserted in $active_4$. The Predictor predicts from it the items **S→•aS@4** (into $predicted_4$) and **S→•@4** (into $completed_4$). The Completer takes the latter item, "reaches out" into $itemset_4$ to find items with the dot in front of an **S**, and finds **S→a•S@3**, which is transformed into **S→aS•@3**, again a completed item. The process then repeats itself two more times, bringing in the items **S→aS•@2** and **S→aS•@1**.

The point to observe here is that, except for the final one all these completed items were temporary results, which cannot be used again by any other action of the parser, and we might as well not store them. This suggests that when a completed item pops up, we should do all the further completions that result from it, and keep only the final results. But there is a snag here: in the above example each completed item led to just one other item, but in the general case a completed item for say a non-terminal A may find more than one item with the dot in front of A, and soon we would be bringing in more and more items. So we restrict our eager completion to completed items whose processing by the Completer results in only one new item; we can then safely discard the original completed item. We keep, however, the original item if it was produced by the Predictor rather than by eager completion, since we will need these items later to construct the parse tree(s). Such a chain of eager completions can stop in one of three ways: 1. the result is a non-completed item; 2. proceeding further would result in more than one item; 3. we cannot proceed further. The last situation can only occur when we have reached the initial item set.

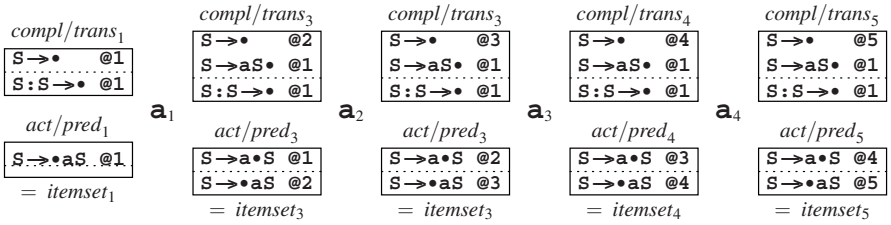
With this space-saving optimization, our parsing looks like:



and indeed the process now requires linear space. Unfortunately it still requires quadratic time: the subsequent Scanner action over $\mathbf{a}_4$ will produce **S→a•S@4**, which produces **S→•@5**, which will make the Completer visit item sets 4 through 1, so the time requirements are still quadratic. The point to notice here is that in doing so the Completer repeated the work it did before on item sets 3 through 1. Once an **S** has been completed in $itemset_4$, the actions described above will be repeated, with the same result: the production of the item **S→aS•@1**, discarding all in-between items. More generally, once an A has been completed with origin position i, the eager completion that follows will always yield the same result. So we can

avoid this waste of effort by recording at each position a list of so called *transitive items*. A transitive item $B : A \rightarrow \alpha\bullet\beta@\,j$ in position i means that if a non-terminal B is recognized in position i, eager completion will yield the item $A \rightarrow \alpha\bullet\beta@\,j$.

With this new item type in place, the Completer completing a B in position i will first check if there is a transitive item for B in that position, and if so, use it. Otherwise it proceeds as described above, doing completion possibly followed by eager completion. If the eager completion results in exactly one item, that item is stored as a transitive item in position i, with the label B. In short, we memoize the result of eager completion provided it is unique. This avoids all duplication of work, and the parser requires linear time on the above and similar grammars:



This improvement was invented by Leo [32], and is much better than one would expect. It can be proved that the modified Earley parser runs in linear time on a very large class of grammars, including the LR(k) grammars for any k (Section 9.4), and even the LR-regular grammars (Section 9.13.2). It should be pointed out, however, that the modified Earley parser has much more overhead than a made-to-measure LR(k) or LR-regular parser. On the other hand, it will avoid duplicate work even on grammars outside these classes, which is of course where its real usefulness lies.

In our zeal to remove items that cannot play a role in recognition, we have also eliminated some items needed for constructing the parse tree. Leo's paper [32] shows how to modify that part of the Earley parser to cope with the deficiencies. It also describes how to handle hidden right recursion.

7.3 Chart Parsing

Chart parsing is not an algorithm but rather a framework in which to develop and experiment with parsers. It can be seen as an abstraction of Earley and CYK parsers, and produces a wide variety of parsers similar to these. It is used extensively in natural language processing, where its flexibility and easy implementability in Prolog are appreciated.

The main data structure in chart parsing is the *chart*, a set of Earley items in our terminology but traditionally interpreted and represented as labeled edges in a graph. This graph has nodes (vertices) between the input tokens (and before and after them); each edge (arc) runs from one vertex to another somewhere on the right of it, or to the same vertex. The nodes are numbered from 1 to $n+1$. Edges are labeled with dotted items; an edge running from node i to node j labeled with a dotted item $A \rightarrow \alpha\bullet\beta$

means that the segment between nodes i and j can be parsed as α and that we hope to be able to extend the edge with a β. We shall write such an edge as $(i, A \rightarrow \alpha \bullet \beta, j)$.

Although there are different ways to treat terminal symbols in chart parsing, it is convenient to make them single productions of non-terminals. A word like "cat" in a position k is then represented by an edge $(k, \texttt{Noun} \rightarrow \texttt{'cat'} \bullet, k+1)$. As a result the algorithms need to handle non-terminals only. This approach also abstracts from the precise value of **Noun** in an early stage.

An edge labeled with an item with the dot in front of a symbol is called an *active edge*, and one with the dot at the end is called an *inactive item*; the terms *passive item* and *completed item* are also used for the latter. Figure 7.26 shows a chart with

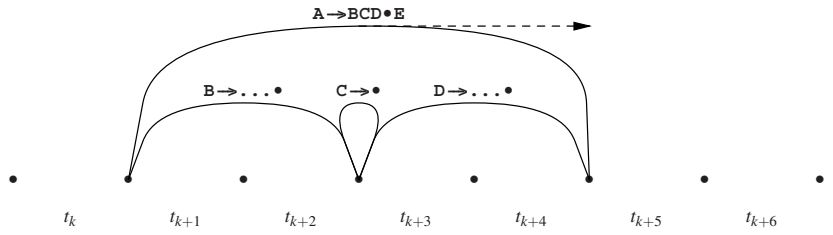


Fig. 7.26. A chart with four edges, three inactive and one active

one active edge, representing the hypothesis **A→BCD•E**, and three inactive ones, representing the fact that **B**, **C**, and **D** have been found. **C** happens to produce ε. The dashed arrow to the right symbolizes the activity of the active edge, looking for an **E**.

7.3.1 Inference Rules

In its most abstract form a chart parsing algorithm is specified by three sets of inference rules, where an inference rule is of the form

If the chart contains edges $E_1, E_2, \cdots$ it must also contain an edge E.

One can say that edge E is required by edges $E_1, E_2, \cdots$; the edges $E_1, E_2, \cdots$ are called the "conditions" and E the "inference".

One set of rules is for completion, another is for steering the parsing process; it can specify top-down, bottom-up, or left-corner parsing, or yet another parsing regime. A third set of inference rules is for initializing the chart. The complete parsing is then defined as the *transitive closure* of the rules over the chart; for transitive closure see Section 3.9. Since the rules in these three sets can easily be changed almost independently, this setup allows great flexibility.

7.3.2 A Transitive Closure Algorithm

The inference mechanism and the transitive closure algorithm are easy to program, but a naive implementation is prone to looping on problematic grammars (those with

loops, left recursion, infinite ambiguity, etc.) We will therefore present a robust implementation of the transitive closure algorithm for chart parsing. In addition to the chart it uses a list of newly discovered edges, the *agenda*; the order in which the edges are kept and retrieved is to be specified later. The basic idea is that at all times the following invariant holds:

> If some edges in the chart and some inference rule require the existence of an edge E, then E is present in the chart and/or the agenda.

This immediately implies that when the agenda is empty, all edges required by edges in the chart are in the chart, and the transitive closure is complete. Also we will have to initialize the chart and agenda so that the invariant already holds. The agenda mechanism is summarized pictorially in Figure 7.27, where the normal arrows indicate information flow and the fat ones represent actions that move edges.

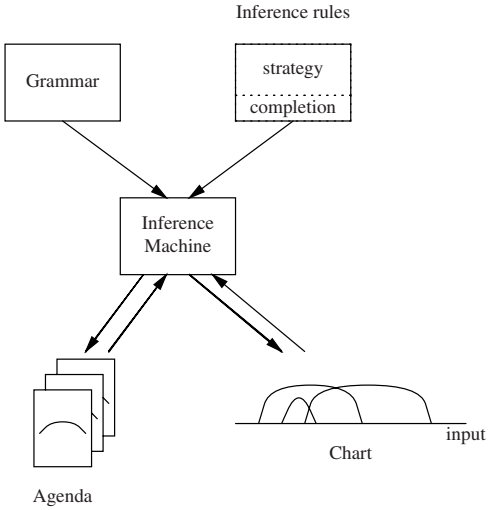


Fig. 7.27. The agenda in chart parsing

The transitive closure algorithm is very simple:

```
until the agenda is empty do:
    extract an edge E from the agenda;
    if E is already in the chart: discard it;
    otherwise:
        apply all inference rules to E and possibly one or more
            edges from the chart, and put the resulting edges,
            if any, in the agenda;
        put E in the chart;
```

Several things should be noted here. First, the algorithm does not specify the order in which edges are obtained from the agenda: the agenda can work as a stack, a first-in-first-out queue, a priority queue, etc. Second, if E is already in the chart, it can

indeed be discarded: all inferences from it and the chart have already been made. Third, the algorithm does not specify the order in which the inference rules are to be applied; the order is immaterial, since the results go into the agenda and do not affect the chart or each other. Fourth, the edge E is put in the chart only *after* all inferences have been drawn from it and put in the agenda, to avoid violating the invariant given above. Fifth, no edge will be placed more than once in the chart; edges may occur multiple times in the agenda, though.

The last, and probably most important thing to note is that the algorithm will always terminate. We can see this as follows. Each cycle of the transitive closure algorithm can do one of two things. It can either remove one edge from the agenda or add a new edge to the chart and possibly add edges to the agenda. Now there are only a finite number of edges possible, so the `otherwise` branch can be taken only a finite number of times. That means that only a finite number of edges can be added to the agenda, and they will eventually all be cleared out by the first action in the loop body. For an example see Section 7.3.6.

7.3.3 Completion

The set of inference rules for completion is the same in almost all chart parsing algorithms, and usually contains only one rule, called the "Fundamental Rule of Chart Parsing"; it says:

> If there is an active edge $(i, A \rightarrow \alpha \bullet B\beta, j)$ and an inactive edge $(j, B \rightarrow \gamma \bullet, k)$, there must be an edge $(i, A \rightarrow \alpha B \bullet \beta, k)$.

Like the Completer in the Earley parser this rule shifts the dot over the B in the item $A \rightarrow \alpha \bullet B\beta$ when we have a completed B.

7.3.4 Bottom-Up (Actually Left-Corner)

We are now in a position to specify a complete chart parsing algorithm. The simplest is probably the algorithm that is usually called "bottom-up chart parsing" but which is actually left-corner. (Pure bottom-up chart parsing is possible but unusual; see Problem 7.6.) It uses only one inference rule:

> If the new edge E has the form $(i, A \rightarrow \alpha \bullet, j)$ (is inactive), add an edge $(i, P \rightarrow A \bullet \beta, j)$ for each rule $P \rightarrow A\beta$ in the grammar.

In other words, upon discovering an A try all rules that have an A as their left corner. For initialization, we leave the chart empty and for all input tokens t_k put $(k, T_k \rightarrow t_k \bullet, k+1)$ in the agenda. If the grammar has ε-rules, we put edges $(k, P \rightarrow \bullet, k)$ for each rule $P \rightarrow \varepsilon$ in the agenda, for all $1 \leq k \leq n+1$. The parser is now ready to run.

7.3.5 The Agenda

The moment we try to run the parser we find that it is underspecified. Which edge should we extract from the agenda first? The simple answer is that it does not matter;

we can extract them in any order and the parser will work. But doing so will recognize the input in a very haphazard way, and if we want a more controlled parsing, we need a more controlled agenda regime.

Suppose that we use the left-corner inference rule; that there are no ε-rules in the grammar; that the agenda acts as a stack; and that the input edges are stacked in reverse order, so the edge for the leftmost token, $(1, T_1 \rightarrow t_1\bullet, 2)$ ends up on top. First the edge for T_1 is retrieved. There are no edges in the chart yet, so the Fundamental Rule does nothing; but the new edge is an inactive edge, so the left-corner rule finds grammar rules that have T_1 for its left corner. In a grammar without ε-rules there must be at least one such rule; otherwise there would be no way to produce a sentence starting with T_1. An edge $(1, A_i \rightarrow T_1\bullet\alpha_i, 2)$ is made for each such rule and pushed on the agenda stack. The edge for T_1 is put in the chart. See Figure 7.28.

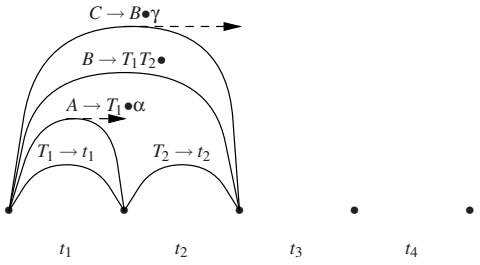


Fig. 7.28. The first few steps in constructing a chart under the left-corner inference rule

In the next cycle the topmost edge of the agenda is retrieved; we will assume it is $(1, A \rightarrow T_1\bullet\alpha, 2)$. Since it is not completed, it is put in the chart, and no new edges are generated. Now the edge for t_2 is on top and is retrieved. It can either activate the Fundamental Rule and combine with the edge for A into a new edge for, say, B, which now spans 2 tokens, or create left-corner edges for itself by the left-corner inference rule, or both. If we assume the edge for B is completed, the left-corner rule will create at least one new edge for it, perhaps for C, starting at position 1. So slowly a left spine is being constructed, as in a left-corner parser.

If there are ε-rules, then, at initialization, edges for them must be put in the agenda stack before the edges for each token, and at the end.

There are many possibilities for the agenda regime. We have seen the stack regime in operation above; it leads to depth-first search, and if the edges for the input tokens are stacked in order, a left-to-right parser results. A queue regime is also possible, resulting in breadth-first search: first all inferences from all tokens are drawn; next all inferences from these inferences are drawn, etc. We already see that this allows left-corner parsing to be used with depth-first search (the usual order) and breadth-first search (very unusual).

Another possibility is to assign a priority to each edge and run the agenda as a priority queue. Not all words in a sentence are equally important, and sometimes it

is wise to start our parsing with the most significant word, which in many languages is the finite — i.e. the conjugated — verb. As a result, the first edge created may already determine the structure of the sentence. Additional inference rules are then required to steer the parsing process so as to obtain edges for the components of this first edge. This is highly useful in head-corner parsing, for which see Section 10.3. A head-corner parser using this technique is described by Proudian and Pollard [198].

7.3.6 Top-Down

A top-down parser tries to predict the next production rule by inspecting the prediction stack. This stack consists of the remainders of right-hand sides that have already been recognized. In our chart parser, if the edge retrieved from the agenda is of the form $(i, A \rightarrow \alpha \bullet B\beta, j)$ (it is an active edge), the prediction it holds is $B\beta$. This leads to the traditional top-down inference rule:

> If the new edge E has the form $(i, A \rightarrow \alpha \bullet B\beta, j)$ (is active), add an edge $(j, B \rightarrow \bullet\gamma, j)$ for each rule $B \rightarrow \gamma$ in the grammar.

In a practical parser this rule will have additional conditions based on look-ahead, but the parser will also work without them. The parser can be initialized by putting an edge for the start symbol running from 1 to 1 and edges for the input tokens on the agenda stack, in that order.

We know that we have to be careful about left recursion in top-down parsers (Section 6.3.2), but it is easy to see that our transitive closure algorithm avoids the problem, as already claimed in Section 7.3.2. Suppose we have a left-recursive non-terminal with the rules `L→La`, `L→Lb`, and `L→c`, and an edge $(i, \texttt{P→...•L...}, j)$ comes up. This causes edges for all three rules for `L` to be pushed in some order, say $(j, \texttt{L→•La}, j)$, $(j, \texttt{L→•c}, j)$, $(j, \texttt{L→•Lb}, j)$; see Figure 7.29(*a*). In this figure we have shown only the items, to fit it on the page; the start

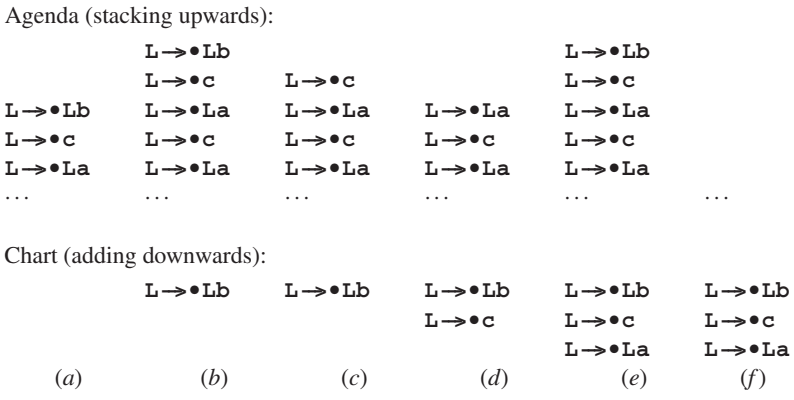


Fig. 7.29. Left recursion: the agenda and the chart

and stop positions are j and j in all cases. Next the edge $(j, \texttt{L}\rightarrow\bullet\texttt{Lb}, j)$ is popped, and causes all three rules for `L` to be pushed again; that done, it is put in the chart (*b*). Now a $(j, \texttt{L}\rightarrow\bullet\texttt{Lb}, j)$ gets popped again, but is discarded since it is already in the chart (*c*). Next an edge $(j, \texttt{L}\rightarrow\bullet\texttt{c}, j)$ is popped, and handed over to the inference mechanism. It may introduce other edges but these are not concerned with the nonterminal `L`; eventually the edge itself is put in the chart (*d*). The edge $(j, \texttt{L}\rightarrow\bullet\texttt{La}, j)$ is popped, and is replaced by another set of three edges for `L`; the edge itself again goes to the chart (*e*). Finally the remaining items are popped one by one, and since all of them are already in the chart, they are all discarded, so the displayed part of the agenda disappears (*f*).

7.3.7 Conclusion

Chart parsing is a very versatile framework for creating and tailoring general context-free parsers, and the present description can only hint at the possibilities. For example, a chart parser will happily accept more than one interpretation of the same token in the input, which is very convenient for natural language parsing. An ambiguous word like "saw" at position k can be entered as two edges, $(k, \texttt{Noun}\rightarrow\texttt{'saw'}\bullet, k+1)$ and $(k, \texttt{VerbPastTense}\rightarrow\texttt{'saw'}\bullet, k+1)$. Chart parsing shares with Earley parsing its $O(n^3)$ time requirements.

Chart parsing was invented by Kay in the early 1970s [16]. Many sophisticated inference rules have been published, for example by Kilbury [24] and Kay [25]. For a comparison of these see Wirén [27]. The agenda mechanism was introduced by Kay [25]. The literature references in (Web)Section 18.1.2 contain many other examples. There are several Prolog implementations of chart parsing on the Internet.

Probably the most extensive application of transitive closure and inference rules to parsing is by Sikkel [158].

Figure 7.30 shows the recognition table of Figure 4.16 in chart format.

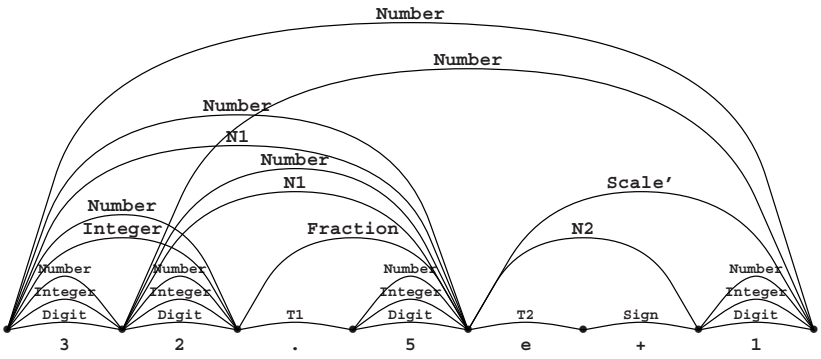


Fig. 7.30. The recognition table of Figure 4.16 in chart format

7.4 Conclusion

General bottom-up parsing methods are powerful because they start from the raw material, the input string, and recognize everything that can be recognized; they leave no stone unturned. Just because of this thoroughness they easily fall prey to exponential explosion. To remedy this, a top-down component is added, resulting in the Earley algorithm. The tabular implementation of this algorithm, chart parsing, allows fine control over the top-down, bottom-up, and prediction content of the parser.

Problems

Problem 7.1: Redesign the depth-first and breadth-first parsing algorithms of Sections 7.1.1 and 7.1.2 so they yield parse-forest grammars (Section 3.7.4) rather than sequences of parse trees.

Problem 7.2: The naive parsing algorithms of Sections 7.1.1 and 7.1.2 do not work for grammars with ε-rules, but intuitively this defect is easily remedied: just recognize ε at all positions and backtrack if the recognition does not lead anywhere. Explain why this plan does not work.

Problem 7.3: Suppose all terminal symbols in a given grammar are different. Can that property be exploited in parser design?

Problem 7.4: The Earley sets from position 1 to a position k contain all prediction stacks possible at k. We can see this as follows. An item $A \rightarrow \alpha\bullet\beta@m$ contains the beginning of a prediction, β. The next segment of the prediction can be found as the δ in the item $X \rightarrow \gamma\bullet A\delta$ in the item set at position m. Such an item must exist, but there may be more than one, in which case the prediction forks, and we get a prediction tree. a) Construct this prediction tree for position 3 in the parsing in Figure 7.11. b) Describe the complete algorithm for constructing the prediction tree at a position k.

Problem 7.5: The transitive items in position i in the improved Earley parser from Section 7.2.5 can be computed right away when $itemset_i$ is constructed, or the computation can be postponed until a Completer comes along for the first time. Comment on the difference.

Problem 7.6: Pure bottom-up chart parsing uses no additional rules besides the Fundamental Rule. Find an initialization that will make this parser work.

Problem 7.7: Show that the left-corner chart parser in this chapter will not loop on grammars with loops in them.

8

Deterministic Top-Down Parsing

In Chapter 6 we discussed two general top-down methods: one using breadth-first search and one using depth-first search. These methods have in common the need to search to find derivations, and thus are not efficient. In this chapter and the next we will concentrate on parsers that do not have to search: there will always be only one possibility to choose from. Parsers with this property are called *deterministic*. Deterministic parsers have several advantages over non-deterministic ones: they are much faster; they produce only one parse tree, so ambiguity is no longer a problem; and this parse tree can be constructed on the fly rather than having to be retrieved afterwards. But there is a penalty: the class of grammars that the deterministic parsing methods are suitable for, while depending on the method chosen, is more restricted than that of the grammars suitable for non-deterministic parsing methods. In particular, only non-ambiguous grammars can be used.

In this chapter we will focus on deterministic top-down methods. As has been explained in Section 3.5.5, there is only one such method, this in contrast with the deterministic bottom-up methods, which will be discussed in the next chapter. From Chapters 3 and 6 we know that in a top-down parser we have a prediction for the rest of the input, and that this prediction has either a terminal symbol in front, in which case we "match", or a non-terminal, in which case we "predict".

It is the predict step that, until now, has caused us so much trouble. The predict step consists of replacing a non-terminal by one of its right-hand sides, and if we have no means to decide which right-hand side to select, we have to try them all. One restriction we could impose on the grammar, one that immediately comes to mind, is limiting the number of alternatives for each non-terminal to one. Then we would need no search, because no selection would be needed. However, such a restriction is far too severe, as it would leave us only with languages that consist of one word. So, limiting the number of right-hand sides per non-terminal to one is not a solution.

There are two sources of information that could help us in selecting the right right-hand side. First there is the partial derivation as it has been constructed so far. However, apart from the prediction this does not give us any information about the rest of the input. The other source of information is the rest of the input. We will

see that looking ahead at the next symbol or the next few symbols will, for certain grammars, tell us which choice to make.

8.1 Replacing Search by Table Look-Up

As a first step we will consider a simple form of grammars which make it particularly easy to limit the search, grammars in which each right-hand side starts with a terminal symbol. In this case, a predict step is always immediately followed by a match step, matching the next input symbol with the symbol starting the right-hand side selected in the prediction. This match step can only succeed for right-hand sides that start with this input symbol. The other right-hand sides will immediately lead to a match step that will fail. We can use this fact to limit the number of predictions as follows: only the right-hand sides that start with a terminal symbol that is equal to the next input symbol will be considered. For example, consider the grammar of Figure 6.1, repeated in Figure 8.1, and the input sentence **aabb**. Using the breadth-first

$$
\begin{array}{lcl}
\mathtt{S}_S & \rightarrow & \mathtt{aB} \mid \mathtt{bA} \\
\mathtt{A} & \rightarrow & \mathtt{a} \mid \mathtt{aS} \mid \mathtt{bAA} \\
\mathtt{B} & \rightarrow & \mathtt{b} \mid \mathtt{bS} \mid \mathtt{aBB}
\end{array}
$$

Fig. 8.1. A grammar producing sentences with an equal number of **a**s and **b**s

top-down method of Chapter 6, extended with the observation described above, results in the steps of Figure 8.2: Frame *a* presents the start of the automaton; we have appended the **#** end marker both to the initial prediction and the input. Only one right-hand side of **S** starts with an **a**, so this is the only applicable right-hand side; this leads to frame *b*. Next, a match step leads to **c**. The next input symbol is again an **a**, so only one right-hand side of **B** is applicable, resulting in frame *d*. Frame *e* is the result of a match step; this time, the next input symbol is a **b**, so two right-hand sides of **B** are applicable; this leads to *f*. Frame *g* is the result of a match step; again, the next input symbol is a **b**, so two right-hand sides of **B** and one right-hand side of **S** are applicable; this leads to frame *h*. This again calls for a match step, leading to *i*. Now there are no applicable right-hand sides for **S** and **A**, because there are no right-hand sides starting with a **#**; thus, these predictions are dead ends. This leaves a match step for the only remaining prediction, leading to frame *j*.

We could enhance the efficiency of this method even further by precomputing the applicable right-hand sides for each non-terminal/terminal combination, and enter these in a table. For the grammar of Figure 8.1, this would result in the table of Figure 8.3. Such a table is called a *parse table* or a *parsing table*.

Despite its title, most of this chapter concerns the construction of these parse tables. Once such a parse table is obtained, the actions of the parser are obvious. The parser does not need the grammar any more. Instead, every time a predict step is called for, the parser uses the next input symbol and the non-terminal at hand as

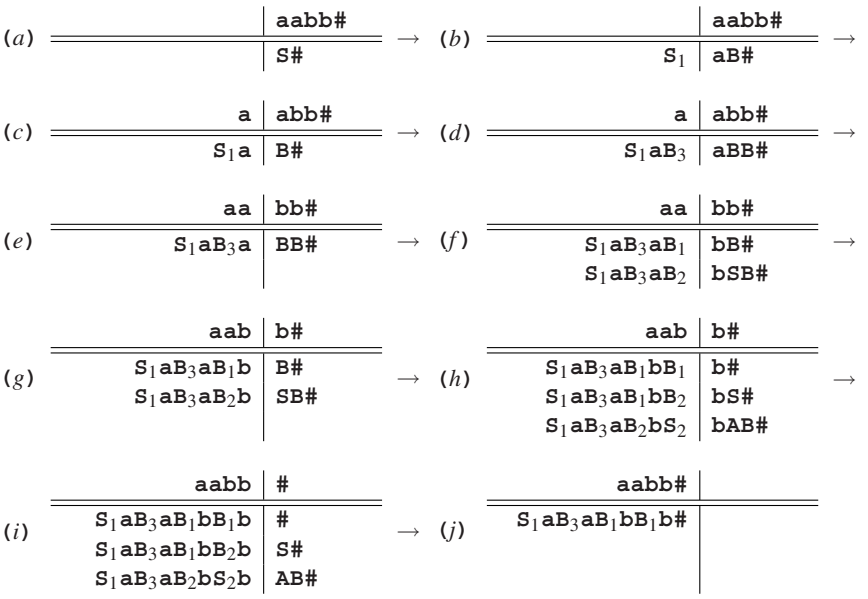


Fig. 8.2. The limited breadth-first parsing of the sentence **aabb**

	a	b	#
S	$S_1 \rightarrow aB$	$S_2 \rightarrow bA$	
A	$A_1 \rightarrow a$	$A_3 \rightarrow bAA$	
	$A_2 \rightarrow aS$		
B	$B_3 \rightarrow aBB$	$B_1 \rightarrow b$	
		$B_2 \rightarrow bS$	

Fig. 8.3. The parse table for the grammar of Figure 8.1

indices in the parse table. The corresponding table entry contains the right-hand sides that have to be considered. For example, in Figure 8.2(*e*), the parser would use input symbol **b** and non-terminal **B** to determine that it has to consider the right-hand sides $\mathbf{B}_1$ and $\mathbf{B}_2$. If the corresponding table entry is empty, we have found an error in the input and the input sentence cannot be derived from the grammar. Using the parse table of Figure 8.3 instead of the grammar of Figure 8.1 for parsing the sentence **aabb** will again lead to Figure 8.2. The advantage of using a parse table is that we do not have to check all right-hand sides of a non-terminal any more to see if they start with the right terminal symbol.

We still have a search process, albeit a more limited one than we had before. Given a prediction *A* and an input token *a*, the search is now confined to the elements of the parse table entry for [*A*,*a*]. This means that we now only need a search because of the [**A**,**a**] and the [**B**,**b**] entries of the table. These entries have more than one element, so we need the search to determine which one results in a derivation of the input sentence.

This last observation is an important one: it immediately leads to a restriction that we could impose on the grammar, to make the parsing deterministic: we could require that each parse table entry contain at most one element. In terms of the grammar, this means that all right-hand sides of a non-terminal start with a different terminal symbol. A grammar which fulfills this requirement is called a *simple LL(1) grammar (SLL(1))*, or an *s-grammar*. "LL(1)" means that the grammar allows a deterministic parser that operates from *L*eft to right, produces a *L*eft-most derivation, using a look-ahead of one (1) symbol.

Consider for example the grammar of Figure 8.4. This grammar generates all

$$\begin{array}{lcl} \mathtt{S}_S & \rightarrow & \mathtt{aB} \\ \mathtt{B} & \rightarrow & \mathtt{b} \mid \mathtt{aBb} \end{array}$$

Fig. 8.4. An example SLL(1) grammar

sentences starting with a number of **a**s, followed by an equal number of **b**s. The grammar is clearly SLL(1). It leads to the parse table of Figure 8.5. The parsing

	a	**b**	**#**
S	$\mathtt{S}_1 \rightarrow \mathtt{aB}$		
B	$\mathtt{B}_2 \rightarrow \mathtt{aBb}$	$\mathtt{B}_1 \rightarrow \mathtt{b}$	

Fig. 8.5. The parse table for the grammar of Figure 8.4

of the sentence **aabb** is presented in Figure 8.6. Again we have added the **#** end marker to signal termination. As expected, there is always only one prediction, so

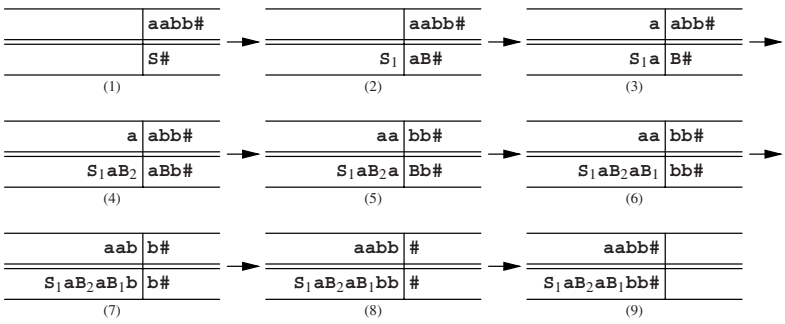

aabb# / S# (1)
aabb# / S1 aB# (2)
a abb# / S1a B# (3)
a abb# / S1aB2 aBb# (4)
aa bb# / S1aB2a Bb# (5)
aa bb# / S1aB2aB1 bb# (6)
aab b# / S1aB2aB1b b# (7)
aabb # / S1aB2aB1bb # (8)
aabb# / S1aB2aB1bb# (9)

Fig. 8.6. The SLL(1) parsing of the sentence **aabb**

no search is needed. Thus, the process is deterministic, and therefore very efficient. The efficiency could be enhanced even further by combining the predict step with the match step that always follows the predict step.

So, SLL(1) grammars lead to simple and very efficient parsers. However, the restrictions that we have placed on the grammar are severe. Not many practical grammars are SLL(1), although many can be transformed into SLL(1) form. In the next section, we will consider a more general class of grammars that still allows the same kind of parser.

8.2 LL(1) Parsing

For the deterministic top-down parser described in the previous section, the crucial restriction placed on the grammar is that all right-hand sides of a non-terminal start with a different terminal symbol. This ensures that each parse table entry contains at most one element. In this section, we will drop the requirement that right-hand sides start with a terminal symbol. We will see that we can still construct a parse table in that case. Later on, we will see that we can even construct a parse table for grammars with ε-rules.

8.2.1 LL(1) Parsing without ε-Rules

If a grammar has no ε-rules, there are no non-terminals that derive the empty string. In other words, each non-terminal ultimately derives strings of terminal symbols of length at least one, and this also holds for each right-hand side. The terminal symbols that start these strings are the ones that we are interested in. Once we know for each right-hand side which terminal symbols can start a string derived from this right-hand side, we can construct a parse table, just as we did in the previous section. So, we have to compute this set of terminal symbols for each right-hand side.

8.2.1.1 FIRST_1 Sets

These sets of terminal symbols are called the "FIRST_1 sets": if we have a non-empty sentential form x, then $\text{FIRST}_1(x)$ is the set of terminal symbols that can start a sentential form derived from x in zero or more production steps. The subscript $_1$ indicates that the set contains single terminal symbols only. Later, we will see FIRST_k sets, consisting of strings of terminal symbols of length at most k. For now, we will drop the subscript $_1$: we will use *FIRST* instead of FIRST_1. If x starts with a terminal symbol, then $\text{FIRST}(x)$ is a set that has this symbol as its only member. If x starts with a non-terminal A, then $\text{FIRST}(x)$ is equal to $\text{FIRST}(A)$, because A cannot produce ε. So, if we can compute the FIRST set for any non-terminal A, we can compute it for any sentential form x. However, $\text{FIRST}(A)$ depends on the right-hand sides of the A-rules: it is the union of the FIRST sets of these right-hand sides. These FIRST sets may again depend on the FIRST set of some non-terminal. This could even be A itself, if the rule is directly or indirectly left-recursive. This observation suggests the iterative process described below to compute the FIRST sets of all non-terminals:

- We first initialize the FIRST sets to the empty set.

- Then we process each grammar rule in the following way: if the right-hand side starts with a terminal symbol, we add this symbol to the FIRST set of the left-hand side, since it can be the first symbol of a sentential form derived from the left-hand side. If the right-hand side starts with a non-terminal symbol, we add all symbols of the present FIRST set of this non-terminal to the FIRST set of the left-hand side. These are all symbols that can be the first terminal symbol of a sentential form derived from the left-hand side.
- The previous step is repeated until no more new symbols are added to any of the FIRST sets.

Eventually, no more new symbols can be added, because the maximum number of elements in a FIRST set is the number of symbols, and the number of FIRST sets is equal to the number of non-terminals. Therefore, the total number of times that a new symbol can be added to any FIRST set is limited by the product of the number of symbols and the number of non-terminals. This is an example of a transitive closure algorithm.

8.2.1.2 Producing the Parse Table

With the help of these FIRST sets, we can now construct a parse table for the grammar. We process each grammar rule $A \rightarrow \alpha$ in the following way: if α starts with a terminal symbol a, we add the right-hand side α to the (A,a) entry of the parse table; if α starts with a non-terminal, we add α to the (A,a) entry of the parse table for all symbols a in FIRST(α). This parse table can then be used for parsing as described in Section 8.1.

Now let us compute the parse table for the example grammar of Figure 8.7. This

```
Session_S  ->  Fact Session
Session_S  ->  Question
Session_S  ->  ( Session ) Session
    Fact   ->  ! STRING
Question   ->  ? STRING
```

Fig. 8.7. An example grammar

grammar describes a simple language that could be used as the input language for a rudimentary consulting system: the user enters some facts, and then asks a question. There is also a facility for sub-sessions. The contents of the facts and questions are of no concern here. They are represented by the word **`STRING`**, which is regarded as a terminal symbol.

We first compute the FIRST sets. Initially, the FIRST sets are all empty. Then, we process all grammar rules in the order of Figure 8.7. The grammar rule **`Session -> Fact Session`** results in adding the symbols from FIRST(**`Fact`**) to FIRST(**`Session`**), but FIRST(**`Fact`**) is still empty. The grammar rule **`Session->Question`** results in adding the symbols from FIRST(**`Question`**)

to FIRST(**Session**), but FIRST(**Question**) is still empty too. The grammar rule **Session → (Session) Session** results in adding **(** to FIRST(**Session**). The grammar rule **Fact → ! STRING** results in adding **!** to FIRST(**Fact**), and the grammar rule **Question → ? STRING** results in adding **?** to FIRST(**Question**). So, after processing all right-hand sides once, we have the following:

FIRST(**Session**)	FIRST(**Fact**)	FIRST(**Question**)
(	**!**	**?**

Next, we process all grammar rules again. This time, the grammar rule **Session → Fact Session** will result in adding **!** (from FIRST(**Fact**)) to FIRST(**Session**), the grammar rule **Session → Question** will result in adding **?** to FIRST(**Session**), and no other changes will take place. So now we get:

FIRST(**Session**)	FIRST(**Fact**)	FIRST(**Question**)
(! ?	**!**	**?**

There were some changes, so we have to repeat this process once more. This time, there are no changes, so the table above presents the FIRST sets of the non-terminals.

Now we have all the information we need to create the parse table. We have to add **Fact Session** to the [**Session**,*a*] entry for all terminal symbols *a* in FIRST(**Fact Session**). The only terminal symbol in FIRST(**Fact Session**) is **!**, so we add **Fact Session** to the [**Session**,**!**] entry. Likewise, we add **Question** to the [**Session**,**?**] entry. Next we add **(Session) Session** to the [**Session**,**(**] entry, **! STRING** to the [**Fact**,**!**] entry, and **? STRING** to the [**Question**,**?**] entry. This results in the parse table of Figure 8.8, where we show just the right-hand sides of the predicted rules in the entries, since the left-hand sides are already shown as the indexes on the left. All parse table entries have at

	!	?	(	)	STRING	#
Session	Fact Session	Question	(Session) Session			
Question		? STRING				
Fact	! STRING					

Fig. 8.8. The parse table for the grammar of Figure 8.7

most one right-hand side, so the parser is deterministic. A grammar without ε-rules is called *LL(1)* if all entries of the parse table, as constructed above, have at most one element, or, in other words, if for every non-terminal *A* the FIRST sets of *A* are pairwise disjoint (no symbol occurs in more than one). If two or more such FIRST sets contain the same symbol, we have a *FIRST/FIRST conflict* and the grammar is not LL(1).

We have lost the S (simplicity) of SLL(1), but the parser is still as simple as before. Producing the parse table has become more difficult, but we have gained

a lot: many practical grammars are LL(1), or are easily transformed into an LL(1) grammar.

8.2.2 LL(1) Parsing with ε-Rules

Not allowing ε-rules is, however, still a major drawback. Certain language constructs are difficult, if not impossible, to describe with an LL(1) grammar without ε-rules. For example, non-terminals that describe lists of terminals or non-terminals are difficult to express without ε-rules. Of course, we could write

```
A → aA | a
```

for a list of **a**s, but this is not LL(1). Compare also the grammar of Figure 8.7 with the one of Figure 8.9. They describe the same language, but the one of Figure 8.9 is much clearer.

```
Session_S → Facts Question | ( Session ) Session
    Facts → Fact Facts | ε
     Fact → ! STRING
 Question → ? STRING
```

Fig. 8.9. The grammar of Figure 8.7 rewritten

8.2.2.1 Extending the FIRST Sets

The main problem with allowing ε-rules is that the FIRST sets, as we have discussed them in the previous section, are not sufficient any more. For example, the **Facts** non-terminal in the grammar of Figure 8.9 has an ε-rule. The FIRST set for this right-hand side is empty, so it does not tell us on which look-ahead symbols we should choose this right-hand side. Also, in the presence of ε-rules, the computation of the FIRST sets itself needs some revision. For example, if we compute the FIRST set of the first right-hand side of **Session** using the method of the previous section, **?** will not be a member, but it should, because **Facts** can derive ε (it is transparent), and then **?** starts a sentential form that can be derived from **Session**.

Let us first extend the FIRST definition to also deal with ε-rules. This time, in addition to terminal symbols, ε will also be allowed as a member of a FIRST set. We will now also have to deal with empty sentential forms, so we will sometimes need the FIRST(ε) set; we will define it as the set containing only the empty string ε. We will also add ε to the FIRST set of a sentential form if this sentential form derives ε.

These may seem minor changes, but the presence of ε-rules affects the computation of the FIRST sets. FIRST($u_1 u_2 \cdots u_n$), which was simply equal to FIRST(u_1), is now computed as follows. We take FIRST(u_1), examine if it contains ε, and if so, we remove the ε and replace it by FIRST($u_2 \cdots u_n$). Apart from this, the computation of the revised FIRST sets proceeds in exactly the same way as before, using the same transitive closure technique.

The treatment of ε is easy to understand: if FIRST(u_1) contains ε, it is transparent, so the tokens in FIRST($u_2 \cdots u_n$) show up through it, and the original ε disappears in the process. Of course the same algorithm is used to compute FIRST($u_2 \cdots u_n$), etc. This chain of events ends at the first u_i whose FIRST(u_i) does not contain ε. If all of the FIRST(u_1), FIRST(u_2), ... FIRST(u_n) contain ε, the last step is the addition of FIRST(ε) to FIRST($u_1 u_2 \cdots u_n$), thus showing that the whole alternative is transparent.

For some algorithms we need to know whether a non-terminal A derives ε. Although we could compute this information separately, using the method described in Section 4.2.1, we can more easily see if ε is a member of the FIRST(A) set as computed. This method uses the fact that if a non-terminal derives ε, ε will ultimately be a member of its FIRST set.

Now let us compute the FIRST sets for the grammar of Figure 8.9. They are first initialized to the empty set. Then, we process each grammar rule: the rule **`Session → Facts Question`** results in adding the terminal symbols from FIRST(**`Facts`**) to FIRST(**`Session`**). However, FIRST(**`Facts`**) is still empty. The rule **`Session → ( Session ) Session`** results in adding **`(`** to FIRST(**`Session`**). Then, the rule **`Facts → Fact Facts`** results in adding the symbols from FIRST(**`Fact`**) (still empty) to FIRST(**`Facts`**), and the rule **`Facts →`** ε results in adding ε to FIRST(**`Facts`**). Then, the rule **`Fact → ! STRING`** results in adding **`!`** to FIRST(**`Fact`**), and the rule **`Question → ? STRING`** adds **`?`** to FIRST(**`Question`**). This completes the first pass over the grammar rules, resulting in:

FIRST(**`Session`**)	FIRST(**`Facts`**)	FIRST(**`Fact`**)	FIRST(**`Question`**)
`(`	ε	**`!`**	**`?`**

The second pass is more interesting: this time, we know that **`Facts`** derives ε, and therefore the rule **`Session → Facts Question`** adds the symbols from FIRST(**`Question`**) (**`?`**) to FIRST(**`Session`**). The rule **`Facts → Fact Facts`** adds **`!`** to FIRST(**`Facts`**). So we get:

FIRST(**`Session`**)	FIRST(**`Facts`**)	FIRST(**`Fact`**)	FIRST(**`Question`**)
`( ?`	ε **`!`**	**`!`**	**`?`**

In the third pass, the only change is the addition of **`!`** to FIRST(**`Session`**), because it is now a member of FIRST(**`Facts`**). So we have:

FIRST(**`Session`**)	FIRST(**`Facts`**)	FIRST(**`Fact`**)	FIRST(**`Question`**)
`( ? !`	ε **`!`**	**`!`**	**`?`**

The fourth pass does not result in any new additions.

The question remains how to decide when an ε right-hand side or, for that matter, a right-hand side which derives ε is to be predicted. Suppose that we have a grammar rule

$$A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n$$

where α_m is or derives ε. Now suppose we find A at the front of a prediction, as in

···	$a \cdots \#$
···	$Ax\#$

where we again have added the # end marker. A breadth-first parser would have to investigate the following predictions:

···	$a \cdots \#$
···	$\alpha_1 x\#$
···	.
···	.
···	$\alpha_n x\#$

We know how to compute the FIRST sets of these predictions, and we know that none of them contains ε, because of the end marker (#). If the next input symbol is not a member of any of these FIRST sets, either the prediction we started with ($Ax\#$) is wrong, or there is an error in the input sentence. Otherwise, the next input symbol is a member of one or more of these FIRST sets, and we can strike out the predictions that do not have the symbol in their FIRST set. Now, if none of these FIRST sets have a symbol in common with any of the other FIRST sets, the next input symbol can only be a member of at most one of these FIRST sets, so at most one prediction remains, and the parser is deterministic at this point.

A context-free grammar is called *LL(1)* if this is always the case. In other words, a grammar is LL(1) if for any prediction $Ax\#$, with A a non-terminal with right-hand sides $\alpha_1, \ldots, \alpha_n$, the sets FIRST($\alpha_1 x\#$), ..., FIRST($\alpha_n x\#$) are pairwise disjoint (no symbol is a member of more than one set). This definition of LL(1) does not conflict with the one that we gave on page 241 for grammars without ε-rules. In that case FIRST($\alpha_i x\#$) is equal to FIRST(α_i) since α_1 is not transparent, so the above definition reduces to the requirement that all FIRST(α_1), ..., FIRST(α_n) be pairwise disjoint.

The above is the official definition of an LL(1) grammar (possibly with ε-rules), but since it requires dynamic computation of FIRST sets, it is usually replaced in practice by a simplified form which allows precomputation at parser generation time. This version is described in the next section. Unfortunately it is often also called "LL(1)", although the official term is "strong-LL(1)"; to avoid confusion we will often use the term "full LL(1)" for the dynamic version described above. Section 8.2.4 gives a reasonably efficient implementation of full LL(1), which allows a large degree of precomputation.

8.2.2.2 The Need for FOLLOW Sets

With the above building blocks we can in principle construct a deterministic parser for any LL(1) grammar. This parser operates by starting with the prediction $S\#$, and its prediction steps consist of replacing the non-terminal at hand with each of its right-hand sides, computing the FIRST sets of the resulting predictions, and checking whether the next input symbol is a member of any of these sets. We then continue with the predictions for which this is the case. If there is more than one prediction, the parser announces that the grammar is not LL(1) and stops.

Although this *is* a deterministic parser, it is far from ideal. First, it does not use a parse table like the one in Figure 8.8, and it does not check the LL(1) property of the grammar until parsing time; we would like to check that property during parser generation time, while computing the parse tables. Second, it is not very efficient, because it has to compute several FIRST sets at each prediction step. We cannot precompute these FIRST sets, because in the presence of ε-rules such a FIRST set depends on all of the predictions (of which there are infinitely many), not just on the first non-terminal. So we still do not know whether, and if so, how we can construct a parse table for an LL(1) grammar with ε-rules, nor do we have a method to determine if such a grammar is LL(1).

Now suppose we have a prediction $Ax\#$ and a rule $A \rightarrow \alpha$, and α is or derives ε. The input symbols that lead to the selection of $A \rightarrow \alpha$ are the symbols in the set FIRST($\alpha x\#$), and as we have seen this set is formed by the symbols in FIRST(α), extended with the symbols in FIRST($x\#$), because of the transparency of α. The set FIRST($x\#$) is the problem: we cannot compute it at parser generation time. What we *can* precompute, though, is the union of all FIRST($x\#$) sets such that $x\#$ can follow A in any prediction. This is just the set of all terminal symbols that can follow A in any sentential form derivable from $S\#$ (not just the present prediction) and is called, quite reasonably, the *FOLLOW* set of A, FOLLOW(A).

It would seem that such a gross approximation would seriously weaken the parser or even make it incorrect, but this is not so. Suppose that this set contains a symbol a that is not a member of FIRST($x\#$), and a is the next input symbol. If a is not a member of FIRST(A), we will predict $A \rightarrow \alpha$, and we will ultimately end up with a failing match, because $\alpha x\#$ does not derive a string starting with an a. So the input string will (correctly) be rejected, although the error will be detected a bit later than before, because the parser may make some ε-predictions before finding out that something is wrong. If a is a member of FIRST(A) then we may have a problem if a is a member of one of the FIRST sets of the other right-hand sides of A. We will worry about this a bit later.

The good thing about FOLLOW sets is that we can compute them at parser generation time. Each non-terminal has a FOLLOW set, and they can be computed as follows:

- as with the computation of the FIRST sets, we start with the FOLLOW sets all empty.
- Next we process all right-hand sides, including the $S\#$ one. Whenever a right-hand side contains a non-terminal, as in $A \rightarrow \cdots By$, we add all symbols from FIRST(y) to FOLLOW(B), since these symbols can follow a B. In addition, if y derives ε, we add all symbols from FOLLOW(A) to FOLLOW(B).
- The previous step is repeated until no more new symbols can be added to any of the FOLLOW sets.

This is again an example of a transitive closure algorithm.

Now let us go back to our example and compute the FOLLOW sets. Starting with **`Session #`**, **`#`** is added to FOLLOW(**`Session`**). Next, the symbols of FIRST(**`Question`**) (**`?`**) are added to FOLLOW(**`Facts`**), because of

the rule **Session → Facts Question**. This rule also adds all symbols of FOLLOW(**Session**) (**#**) to FOLLOW(**Question**).

The rule **Session → (Session) Session** results in adding the **)** symbol to FOLLOW(**Session**) and in a spurious addition of all symbols of FOLLOW(**Session**) to FOLLOW(**Session**). The next rule is the rule **Facts → Fact Facts**. All symbols from FIRST(**Facts**) (**!**) are added to FOLLOW(**Fact**), and since **Facts** produces empty, all symbols from FOLLOW(**Facts**) (**?**) are added to FOLLOW(**Fact**). The other rules do not result in any additions. So, after the first pass we have:

FOLLOW(**Session**)	FOLLOW(**Facts**)	FOLLOW(**Fact**)	FOLLOW(**Question**)
#)	**?**	**! ?**	**#**

In the second pass, **)** is added to FOLLOW(**Question**), because it is now a member of FOLLOW(**Session**), and all members of FOLLOW(**Session**) become a member of FOLLOW(**Question**) because of the rule **Session → Facts Question**.

In the third pass no changes take place. The resulting FOLLOW sets are presented below:

FOLLOW(**Session**)	FOLLOW(**Facts**)	FOLLOW(**Fact**)	FOLLOW(**Question**)
#)	**?**	**! ?**	**#)**

8.2.2.3 Using the FOLLOW Sets to Produce a Parse Table

Once we know the FOLLOW set for each non-terminal that derives ε, we can construct a parse table. First we compute the FIRST set of each non-terminal. This also tells us which non-terminals derive ε. Next, we compute the FOLLOW set of each non-terminal. Then, starting with an empty parse table, we process each grammar rule $A \rightarrow \alpha$ as follows: we add the right-hand side α to the $[A,a]$ entry of the parse table for all terminal symbols a in FIRST(α), as we did before. This time however, we also add α to the $[A,a]$ entry of the parse table for all terminal symbols a in FOLLOW(A) when α is or derives ε (when FIRST(α) contains ε). A shorter way of saying this is that we add α to the $[A,a]$ entry of the parse table for all terminal symbols a in FIRST(α FOLLOW(A)). This last set consists of the union of the FIRST sets of the sentential forms αb for all symbols b in FOLLOW(A).

If a token in a FOLLOW set causes the addition of a right-hand side to an entry that already contains a right-hand side due to a token in a FIRST set, we have a *FIRST/FOLLOW conflict*, and the grammar is not LL(1). It is even possible to have a *FOLLOW/FOLLOW conflict*: an entry receives two right-hand sides, both brought in by tokens from FOLLOW sets. This happens if more than one alternative of a non-terminal can produce ε.

Now let us produce a parse table for our example. The **Session → Facts Question** rule does not derive ε, because **Question** does not. Therefore, only the terminal symbols in FIRST(**Facts Question**) lead to addition of this rule to the table. FIRST(**Facts Question**) contains **!** from

FIRST(**Facts**) and **?** from FIRST(**Question**) because **Facts** derives ε. So the right-hand side **Facts Question** must be entered in the entries [**Session**,**!**] and [**Session**,**?**]; the right-hand side **(Session) Session** should be added to entry [**Session**,**(**].

FIRST(**Fact Facts**) is {**!**}, so this right-hand side is entered in [**Facts**,**!**]. Since the right-hand side of **Facts**→ε produces ε but has an otherwise empty FIRST set, its look-ahead set is FOLLOW(**Facts**), which contains just **?**; so the right-hand side ε is entered in entry [**Facts**,**?**].

Similarly, all other rules are added, resulting in the parse table presented in Figure 8.10.

	(	)	#	!	?	STRING
Session	(Session) Session			Facts Question	Facts Question	
Facts				Fact Facts	ε	
Fact				! STRING		
Question					? STRING	

Fig. 8.10. The parse table for the grammar of Figure 8.9

8.2.3 LL(1) versus Strong-LL(1)

If all entries of the resulting parse table have at most one element, the parser is again deterministic. In this case, the grammar is called *strong-LL(1)* and the parser is called a strong-LL(1) parser. In the literature, strong-LL(1) is often referred to as "strong LL(1)", without a hyphen between the words "strong" and "LL". This is misleading because it indicates that "strong" belongs to "grammar" rather than to "LL(1)", which in turn suggests that the class of strong-LL(1) grammars is more powerful than the class of LL(1) grammars. This is not the case: every strong-LL(1) grammar is LL(1).

It is perhaps more surprising that every LL(1) grammar is strong-LL(1). In other words, every grammar that is not strong-LL(1) is not LL(1), and this is demonstrated with the following argument: if a grammar is not strong-LL(1), there is a parse table entry, say (*A*,*a*), with at least two elements, say α and β. This means that *a* is a member of both FIRST(α FOLLOW(*A*)) and FIRST(β FOLLOW(*A*)). Now there are three possibilities:

- *a* is a member of both FIRST(α) and FIRST(β). In this case, the grammar cannot be LL(1), because for any prediction *Ax***#**, *a* is a member of both FIRST(α*x*#) and FIRST(β*x*#).
- *a* is a member of either FIRST(α) or FIRST(β), but not both. Let us assume, without loss of generality, that *a* is a member of FIRST(α). In this case, *a* is still a member of FIRST(β FOLLOW(*A*)), so there is a prediction *Ax***#**, such that *a* is a member of FIRST(β*x***#**). However, *a* is also a member of FIRST(α*x***#**), so the

grammar is not LL(1). In other words, in this case there is a prediction in which an LL(1) parser cannot decide which right-hand side to choose either.

- *a* is neither a member of FIRST(α), nor a member of FIRST(β). In this case α and β must derive ε and *a* must be a member of FOLLOW(*A*). This means that there is a prediction *Ax*# such that *a* is a member of FIRST(*x*#) and thus *a* is a member of both FIRST(α*x*#) and FIRST(β*x*#), so the grammar is not LL(1). This means that in an LL(1) grammar at most one right-hand side of any non-terminal derives ε.

8.2.4 Full LL(1) Parsing

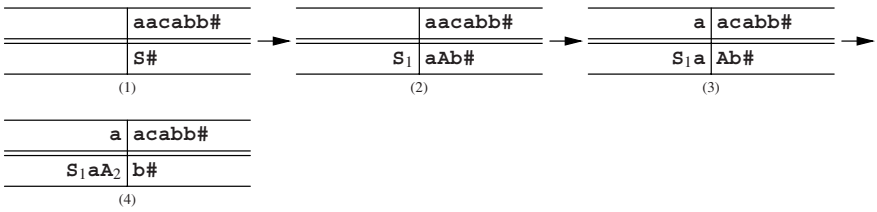
We already mentioned briefly that an important difference between LL(1) parsing and strong-LL(1) parsing is that the strong-LL(1) parser sometimes makes ε-predictions before detecting an error. Consider for example the following grammar:

$$\begin{array}{lcl} \mathtt{S}_S & \rightarrow & \mathtt{a\ A\ b} \mid \mathtt{b\ A\ a} \\ \mathtt{A} & \rightarrow & \mathtt{c\ S} \mid \varepsilon \end{array}$$

The strong-LL(1) parse table of this grammar is:

	a	b	c	#
S	a A b	b A a		
A	ε	ε	c S	

Now, on input sentence **aacabb**, the strong-LL(1) parser makes the following moves:

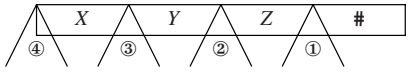


The problem here is that the prediction is destroyed by the time the error is detected. In contrast, a full-LL(1) parser would not do the last step, because neither FIRST(**b#**), nor FIRST(**cSb#**) contain **a**, so the full-LL(1) parser would detect the error before choosing a right-hand side for **A**. A full-LL(1) parser has the *immediate error detection property*, which means that an error is detected as soon as the erroneous symbol is first examined, whereas a strong-LL(1) parser only has the *correct-prefix property*, which means that the parser detects an error as soon as an attempt is made to match (or shift) the erroneous symbol. In Chapter 16, we will see that the immediate error detection property will help improve error recovery.

Given a prediction $A \cdots$ #, a full-LL(1) parser bases its parsing decisions on FIRST($A \cdots$ #) rather than on the approximation FIRST(*A* FOLLOW(*A*)); this avoids any parsing decisions on erroneous input symbols (which can never occur in

FIRST($A \cdots$ **#**) but may occur in FIRST(A FOLLOW(A))). So, if we have prediction $A \cdots$ **#** and input symbol a, we first have to determine if a is a member of FIRST($A \cdots$ **#**), before consulting the parse table to choose a right-hand side for A. The penalty for this is in efficiency: every time that parse table has to be consulted, a FIRST set has to be computed and a check made that the input symbol is a member.

Fortunately, we can do better than this. A first step to improvement is the following: suppose that we maintain between all symbols in the prediction a set of terminal symbols that are correct at this point, like this:

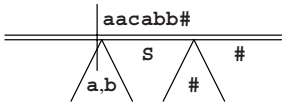


Here, ① is the set of symbols that are legal at this point; this is just the FIRST set of the remaining part of the prediction: FIRST(**#**); likewise, ② is FIRST(Z**#**), ③ is FIRST(YZ**#**), and ④ is FIRST(XYZ**#**). These sets can easily be computed, from right to left. For example, ③ consists of the symbols in FIRST(Y), with the symbols from ② added if Y derives ε (if ε is a member of FIRST(Y)). When a non-terminal is replaced by one of its right-hand sides, the set behind this right-hand side is available, and we can use this to compute the sets within this right-hand side and in front of it. Since none of these sets contain ε, they give an immediate answer to the question which prediction to choose.

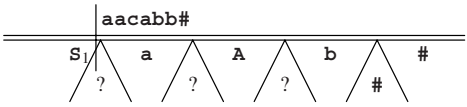
Now let us see how this works for our example. As the reader can easily verify,

FIRST(**S**) = { **a**, **b**}, and
FIRST(**A**) = { **c**, ε}.

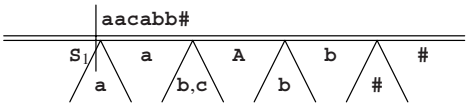
The parser starts with the prediction **S#**. We have to find a starting point for the sets: it makes sense to start with an empty one to the right of the **#**, because no symbols are correct after the **#**. So the parser starts in the following state:



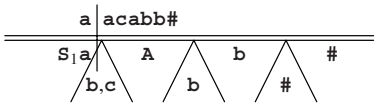
The first input symbol is a member of the current FIRST set, so it is correct. The (**S**, **a**) entry of the parse table contains **aAb**, so we get parser state



Computing the sets marked with a question mark from right to left results in the following parser state:



Note that **b** now is a member of the set in front of **A**, but **a** is not, although it is a member of FOLLOW(**A**). After the match step, the parser is in the following state:



The next input symbol is not a member of the current FIRST set, so an error is detected, and no right-hand side of **A** is chosen. Instead, the prediction is left intact, so error recovery can profit from it.

It is not clear that all this is more efficient than computing the FIRST set of a prediction to determine the correctness of an input symbol before choosing a right-hand side. However, it does suggest that we can do this at parser generation time, by combining non-terminals with the FIRST sets that can follow it in a prediction. For our example, we always start with non-terminal **S** and the set {**#**}. We will indicate this with the pair [**S**,{**#**}]. Starting with this pair, we will try to make rules for the behavior of each pair that turns up, for each valid look-ahead. We know from the FIRST sets of the alternatives for **S** that on look-ahead symbol **a**, [**S**,{**#**}] results in right-hand side **aAb**. Now the only symbol that can follow **A** here is a **b**. So in fact, we have:

on look-ahead symbol **a**, [**S**,{**#**}] results in right-hand side **a** [**A**,{**b**}] **b**.

Similarly we find:

on look-ahead symbol **b**, [**S**,{**#**}] results in right-hand side **b** [**A**,{**a**}] **a**.

We have now obtained pairs for **A** followed by a **b**, and **A** followed by an **a**. So we have to make rules for them: We know that on look-ahead symbol **c**, [**A**,{**b**}] results in right-hand side **cS**. Because **A** can only be followed by a **b** in this context, the same holds for this **S**. This gives:

on look-ahead symbol **c**, [**A**,{**b**}] results in right-hand side **c** [**S**,{**b**}].

Likewise, we get the following rules:

on look-ahead symbol **b**, [**A**,{**b**}] results in right-hand side ε;
on look-ahead symbol **c**, [**A**,{**a**}] results in right-hand side **c** [**S**,{**a**}];
on look-ahead symbol **a**, [**A**,{**a**}] results in right-hand side ε.

Now we have to make rules for the pairs **S** followed by an **a**, and **S** followed by a **b**:

on look-ahead symbol **a**, [**S**,{**a**}] results in right-hand side **a** [**A**,{**b**}] **b**;
on look-ahead symbol **b**, [**S**,{**a**}] results in right-hand side **b** [**A**,{**a**}] **a**;

on look-ahead symbol **a**, [**S**,{**b**}] results in right-hand side **a** [**A**,{**b**}] **b**;
on look-ahead symbol **b**, [**S**,{**b**}] results in right-hand side **b** [**A**,{**a**}] **a**.

In fact, we find that we have rewritten the grammar, using the (non-terminal, followed-by set) pairs as non-terminals, into the following form:

```
[S,{#}]  →  a [A,{b}] b | b [A,{a}] a
[S,{a}]  →  a [A,{b}] b | b [A,{a}] a
[S,{b}]  →  a [A,{b}] b | b [A,{a}] a
[A,{a}]  →  c [S,{a}] | ε
[A,{b}]  →  c [S,{b}] | ε
```

For this grammar, the following parse table can be produced:

	a	b	c	#
[S,{#}]	a [A,{b}] b	b [A,{a}] a		
[S,{a}]	a [A,{b}] b	b [A,{a}] a		
[S,{b}]	a [A,{b}] b	b [A,{a}] a		
[A,{a}]	ε		c [S,{a}]	
[A,{b}]		ε	c [S,{b}]	

The entries for the different [**S**,...] rules are identical so we can merge them. After that, the only change with respect to the original parse table is the duplication of the **A**-rule: now there is one copy for each context in which **A** has a different set behind it in a prediction.

Now, after accepting the first **a** of **aacabb**, the prediction is [**A**,{**b**}]**b#**; since the parse table entry ([**A**,{**b**}], **a**) is empty, parsing will stop here and now.

The resulting parser is exactly the same as the strong-LL(1) one. Only the parse table is different. Often, the LL(1) table is much larger than the strong-LL(1) one. As the benefit of having an LL(1) parser only lies in that it detects some errors a bit earlier, this usually is not considered worth the extra cost, and thus most parsers that are advertised as LL(1) parsers are actually strong-LL(1) parsers.

In summary, confronted with a prediction stack $A\alpha$ and a grammar rule $A \rightarrow \beta$,

- a (full) LL(1) parser bases its decisions on the FIRST set of $\beta\alpha$ and the first token of the input;
- a strong-LL(1) parser bases its decisions on the FIRST set of β, the FOLLOW set of A when β produces ε, and the first token of the input;
- a simple-LL(1) parser bases its decisions on the first token of β and the first token of the input.

8.2.5 Solving LL(1) Conflicts

If a parse table entry has more than one element, we have an "*LL(1) conflict*". In this section, we will discuss how to deal with them. We have already seen one way to deal with conflicts: use a depth-first or a breadth-first parser with a one symbol look-ahead. This, however, has several disadvantages: the resulting parser is not deterministic any more, it is less efficient (often to such an extent that it becomes

unacceptable), and it still does not work for left-recursive grammars. Therefore, we want to try and eliminate these conflicts, so we can use an ordinary LL(1) parser.

Two techniques that can help us here are left recursion elimination and left-factoring. These can be performed by hand relatively easily and are described in the next two sections. Grammars for which these two techniques are sufficient are called "kind grammars"; see Žemlička and Král [106, 107, 108] for their precise definition and processing.

8.2.5.1 Left-Recursion Elimination

The first step to take is the elimination of left recursion. Left-recursive grammars always lead to LL(1) conflicts, because the right-hand side causing the left recursion has a FIRST set that contains all symbols from the FIRST set of the non-terminal. Therefore, it also contains all terminal symbols of the FIRST sets of the other right-hand sides of the non-terminal. Eliminating left recursion has already been discussed in Section 6.4.

8.2.5.2 Left-Factoring

A further technique for removing LL(1) conflicts is *left-factoring*. Left-factoring of grammar rules is like factoring arithmetic expressions:

$$a \times b + a \times c = a \times (b + c).$$

The grammatical equivalent to this is a rule

$$A \rightarrow xy \mid xz$$

which clearly has an LL(1) conflict on the terminal symbols in FIRST(x). We replace this grammar rule with the two rules

$$A \rightarrow xN$$
$$N \rightarrow y \mid z$$

where N is a new non-terminal. There have been some attempts to automate this process; see Foster [405], Hammer [406], and Rosenkrantz and Hunt [408].

8.2.5.3 Conflict Resolvers

Sometimes, these techniques do not help much. We could for example be dealing with a language for which no LL(1) grammar exists. In fact, many languages can be described by a context-free grammar, but not by an LL(1) grammar. Another method of handling conflicts is to resolve them by so-called *disambiguating* rules. An example of such a disambiguating rule is: "on a conflict, the textually first of the conflicting right-hand sides is chosen". With this disambiguating rule, the order of the right-hand sides within a grammar rule becomes crucial, and unexpected results may occur if the grammar-processing program does not clearly indicate where conflicts occur and how they are resolved.

A better method is to have the grammar writer specify explicitly how each conflict must be resolved, using so-called *conflict resolvers*. One option is to resolve conflicts at parser generation time. Parser generators that allow this kind of conflict resolver usually have a mechanism that enables the user to indicate (at parser generation time) which right-hand side must be chosen on a conflict. Another, much more flexible method is to have conflicts resolved at parse time. When the parser meets a conflict, it calls a user-specified conflict resolver. Such a conflict resolver has the complete left-context at its disposal, so it could base its choice on this left context. It is also possible to have the parser look further ahead in the input, and then resolve the conflict based on the symbols found. See Milton, Kirchhoff and Rowland [337] and Grune and Jacobs [362], for similar approaches using attribute grammars. (Attribute grammars are discussed in Section 15.3.1.)

8.2.6 LL(1) and Recursive Descent

Most hand-written parsers are LL(1) parsers. They usually are written in the form of a non-backtracking compiled recursive-descent parser (see Section 6.6). In fact, this is a very simple way to implement a strong-LL(1) parser. For a non-terminal A with grammar rule

$$A \rightarrow \alpha_1 \mid \cdots \mid \alpha_n$$

the parsing routine has the following structure:

```
procedure A;
    if look_ahead ∈ FIRST(α1 FOLLOW(A)) then
        code for α1 ...
    else if look_ahead ∈ FIRST(α2 FOLLOW(A)) then
        code for α2 ...
        :
    else if look_ahead ∈ FIRST(αn FOLLOW(A)) then
        code for αn ...
    else ERROR;
end A;
```

The look-ahead symbol always resides in a variable called "look_ahead". The procedure ERROR announces an error and stops the parser.

The code for a right-hand side consists of the code for the symbols of the right-hand side. A non-terminal symbol results in a call to the parsing routine for this non-terminal, and a terminal symbol results in a call to a MATCH routine with this symbol as parameter. This MATCH routine has the following structure:

```
procedure MATCH(sym);
    if look_ahead = sym then
        look_ahead := NEXTSYM
    else ERROR;
end MATCH;
```

The NEXTSYM procedure reads the next symbol from the input.

Several LL(1) parser generators produce a recursive descent parser instead of a parse table that is to be interpreted by a grammar-independent parser. The advantages of generating a recursive descent parser are numerous:

- Semantic actions are easily embedded in the parsing routines.
- A parameter mechanism or attribute mechanism comes virtually for free: the parser generator can use the parameter mechanism of the implementation language.
- Non-backtracking recursive descent parsers are quite efficient, often more efficient than the table-driven ones.
- Dynamic conflict resolvers are implemented easily.

The most important disadvantage of generating a recursive descent parser is the size of the parser. A recursive descent parser is usually larger than a table-driven one (including the table). With present computer memories this is no longer a problem, however.

8.3 Increasing the Power of Deterministic LL Parsing

There are many situations in which a look-ahead of one token is not enough. A prime example is the definition of an element of an expression in a programming language:

```
element_S  →  idf | idf ( parameters ) | idf [ indexes ]
```

where **idf** produces identifiers. This grammar fragment defines expression elements like **x**, **sin(0.41)**, and **T[3,1]**, each of which starts with an identifier; only the second token allows us to distinguish between the alternatives.

There are several ways to increase the power of deterministic LL parsing, and we have already seen one above: conflict resolvers. This section concentrates on extending the look-ahead, first to a bounded number of tokens and then to an unbounded number. In between we treat an efficient compromise.

8.3.1 LL(k) Grammars

It is possible and occasionally useful to have a look-ahead of k symbols with $k > 1$, leading to LL(k) grammars. To achieve this, we need a definition of FIRST_k sets: if x is a sentential form, then $FIRST_k(x)$ is the set of terminal strings w such that $|w|$ (the length of w) is less than k and $x \xrightarrow{*} w$, or $|w|$ is equal to k, and $x \xrightarrow{*} wy$, for some sentential form y. For $k = 1$ this definition coincides with the definition of the FIRST sets as we have seen it before.

We now have the instruments needed to define LL(k): a grammar is *LL(k)* if for any prediction $Ax\#^k$, with A a non-terminal with right-hand sides $\alpha_1, \ldots, \alpha_n$, the sets $\text{FIRST}_k(\alpha_1 x\#^k), \ldots, \text{FIRST}_k(\alpha_n x\#^k)$ are pairwise disjoint. (Here $\#^k$ represents a sequence of k #s; they are required to supply enough look-ahead tokens for checking near the end of the input string.) Obviously, for any k, the set of LL(k) grammars is

a subset of the set of LL($k+1$) grammars, and in fact, for any k there are LL($k+1$) grammars that are not LL(k). A trivial example of this is given in Figure 8.11. Less

$$\mathtt{S_S} \;\rightarrow\; \mathtt{a}^k\mathtt{b} \;\mid\; \mathtt{a}^k\mathtt{a}$$

Fig. 8.11. An LL($k+1$) grammar that is not LL(k)

obvious is that for any k there are languages that are LL($k+1$), but not LL(k). An example of such a language is given in Figure 8.12. See Kurki-Suonio [42] for more

$$\begin{array}{lcl} \mathtt{S_S} & \rightarrow & \mathtt{aSA} \;\mid\; \varepsilon \\ \mathtt{A} & \rightarrow & \mathtt{a}^k\mathtt{bS} \;\mid\; \mathtt{c} \end{array}$$

Fig. 8.12. A grammar defining an LL($k+1$) language that is not LL(k)

details.

With LL(k) grammars we have the same problem as with the LL(1) grammars: producing a parse table is difficult. In the LL(1) case, we solved this problem with the aid of the FOLLOW sets, obtaining strong-LL(1) parsers. We can try the same with LL(k) grammars using FOLLOW$_k$ sets. For any non-terminal A, *FOLLOW$_k$(A)* is now defined as the union of the sets FIRST$_k(x\#^k)$, for any prediction $Ax\#^k$.

Once we have the FIRST$_k$ sets and the FOLLOW$_k$ sets, we can produce a parse table for the grammar. Like the LL(1) parse table, this parse table will be indexed with pairs consisting of a non-terminal and a terminal string of length equal to k. Every grammar rule $A \rightarrow \alpha$ is processed as follows: α is added to the (A, w) entry of the table for every w in FIRST$_k(\alpha$ FOLLOW$_k(A))$ (as we have seen before, this last set denotes the union of several FIRST$_k$ sets: it is the union of all FIRST$_k(\alpha v)$ sets with v an element of FOLLOW$_k(A)$). All this is just an extension to k look-ahead symbols of what we did earlier with one look-ahead symbol.

If this results in a parse table where all entries have at most one element, the grammar is *strong-LL(k)*. Unlike the LL(1) case however, for $k > 1$ there are grammars that are LL(k), but not strong-LL(k). An example of such a grammar is given in Figure 8.13.

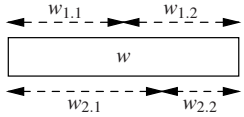
$$\begin{array}{lcl} \mathtt{S_S} & \rightarrow & \mathtt{aAaa} \;\mid\; \mathtt{bAba} \\ \mathtt{A} & \rightarrow & \mathtt{b} \;\mid\; \varepsilon \end{array}$$

Fig. 8.13. An LL(2) grammar that is not strong-LL(2)

This raises an interesting question, one that has kept the authors busy for quite a while: why is it different for $k = 1$? If we try to repeat our proof from Section 8.2.3 for a look-ahead $k > 1$, we see that we fail at the very last step: let us examine a strong-LL(k) conflict: suppose that the right-hand sides α and β both end up in the

(A, w) entry of the parse table. This means that w is a member of both $\text{FIRST}_k(\alpha\ \text{FOLLOW}_k(A))$ and $\text{FIRST}_k(\beta\ \text{FOLLOW}_k(A))$. Now there are three cases:

- w is a member of both $\text{FIRST}_k(\alpha)$ and $\text{FIRST}_k(\beta)$. In this case, the grammar cannot be LL(k), because for any prediction $Ax\#^k$, w is a member of both $\text{FIRST}_k(\alpha x\#^k)$ and $\text{FIRST}_k(\beta x\#^k)$.
- w is a member of either $\text{FIRST}_k(\alpha)$ or $\text{FIRST}_k(\beta)$, but not both. Let us say that w is a member of $\text{FIRST}_k(\alpha)$. In this case, w still is a member of $\text{FIRST}_k(\beta\ \text{FOLLOW}_k(A))$ so there is a prediction $Ax\#^k$, such that w is a member of $\text{FIRST}_k(\beta x\#^k)$. However, w is also a member of $\text{FIRST}_k(\alpha x\#^k)$, so the grammar is not LL(k). In other words, in this case there is a prediction in which an LL(k) parser cannot decide which right-hand side to choose either.
- w is neither a member of $\text{FIRST}_k(\alpha)$ nor a member of $\text{FIRST}_k(\beta)$. Here, we have to deviate from the reasoning we used in the LL(1) case. As w is an element of $\text{FIRST}_k(\alpha\ \text{FOLLOW}_k(A))$, w can now be split into two parts $w_{1.1}$ and $w_{1.2}$, such that $w_{1.1}$ is an element of $\text{FIRST}_k(\alpha)$ and $w_{1.2}$ is a non-empty start of an element of $\text{FOLLOW}_k(A)$. Likewise, w can be split into two parts $w_{2.1}$ and $w_{2.2}$ such that $w_{2.1}$ is an element of $\text{FIRST}_k(\beta)$ and $w_{2.2}$ is a non-empty start of an element of $\text{FOLLOW}_k(A)$. So we have the following situation:



  Now, if $w_{1.1} = w_{2.1}$, $w_{1.1}$ is a member of $\text{FIRST}_k(\alpha)$, as well as $\text{FIRST}_k(\beta)$, and there is a prediction $Ax\#^k$ such that $x\#^k \gtrdot^* w_{1.2} \cdots$. So $\text{FIRST}_k(\alpha x\#^k)$ contains w and so does $\text{FIRST}_k(\beta x\#^k)$, and therefore, the grammar is not LL(k). So the only case left is that $w_{1.1} \neq w_{2.1}$. Neither $w_{1.2}$ nor $w_{2.2}$ are ε, and this is just impossible if $|w| = 1$.

Strong-LL(k) parsers with $k > 1$ are seldom used in practice, partly because the gain is marginal and the same effect can often be obtained by using conflict resolvers, and partly because the parse tables can be large. That problem may, however, have been exaggerated in the literature, since the table entries are mostly empty and the tables lend themselves very well to table compression.

To obtain a full-LL(k) parser, the method that we used to obtain a full-LL(1) parser can be extended to deal with pairs (A, L), where L is a FIRST_k set of $x\#^k$ in some prediction $Ax\#^k$. This extension is straightforward and will not be discussed further.

8.3.2 Linear-Approximate LL(k)

The large LL(k) tables and their heavy construction mechanism can often be avoided by a simple trick: in addition to FIRST sets of non-terminals and alternatives, we introduce *SECOND sets*: the set of tokens that can occur in second position in the

terminals productions of a non-terminal or alternative. We then choose an alternative A if the first token of the input is in FIRST(A) and the second token of the input is in SECOND(A). Rather than a table of size $O(t^2)$, where t is the number of terminals in the grammar, this technique needs two tables of size $O(t)$; also, the tables are easier to generate. This gives us a poor man's version of LL(2), called "linear-approximate LL(2)", and, with the introduction of THIRD, FOURTH, etc. sets, *linear-approximate LL(k)*.

If the first non-terminal in an alternative produces only one token, its FOLLOW set must be called upon to create the correct SECOND set for that alternative. With this provision, it is easy to see that linear-approximate LL(2) efficiently and cheaply handles the grammar fragment for expression elements at the beginning of Section 8.3 on page 254.

In principle linear-approximate LL(2) is weaker than LL(2), because it breaks the relationships between the two tokens in the look-ahead sets. If two alternatives in an LL(2) parser have look-ahead sets of $\{ab, cd\}$ and $\{ad, cb\}$ respectively, they are disjoint; but under linear-approximate LL(2) both have a FIRST set of ac and a SECOND set of bd, so they are no longer disjoint. In practice this effect is rare, though.

Linear-approximate LL was first described by Parr and Quong [51], who also give implementation details.

8.3.3 LL-Regular

LL(k) provides bounded look-ahead, but grammar rules like `A->Bb | Bc` with `B` producing for example `a`* show that bounded look-ahead will not always suffice: the discriminating token can be arbitrarily far away.

This suggests unbounded look-ahead, but that is easier said than done. Unbounded look-ahead is much more important and has been investigated much more extensively in LR parsing, and is treated in depth in Section 9.13.2. We will give here just an outline of the LL version; for details see Jarzabek and Krawczyk [44], Nijholt [45], Poplawski [47], and Nijholt [48].

If bounded look-ahead is not enough, we need a way to describe the set of unbounded look-ahead sequences, which suggests a grammar. And indeed it turns out that each alternative defines its own context-free look-ahead grammar. But this has two problems: it solves the parsing problem by almost doing the same parsing, and we cannot decide if the look-ahead grammars of two alternatives are disjoint. To solve both problems we approximate the CF grammars by regular grammars (hence the term "LL-regular"): disjointness of regular expressions can be decided, and there is a trick to do regular analysis of the input only once for the entire parsing.

There is no hard and fast algorithm for the approximation of the CF grammars with regular grammars, but there are many heuristics.

LL-regular is probably of theoretical interest only; if a parser writer goes through that much trouble, the effort is more wisely spent on LR-regular. Still, it offers many interesting insights, and the parser landscape would be incomplete without it.

8.4 Getting a Parse Tree Grammar from LL(1) Parsing

Getting a parse tree grammar from LL(1) parsing is straightforward. The basic idea is to create a new grammar rule for each prediction; the non-terminals in the right-hand side are numbered using an increasing global counter, and the resulting right-hand side is also inserted in the prediction stack. This produces numbered non-terminals in the prediction stack, which then lead to the creation of more newly numbered rules. The created rules then form the parse tree grammar. Since the parser is deterministic, there is only one parse, and we obtain a parse *tree* grammar rather than a parse forest grammar.

To see how it works in some more detail we refer to the grammar in Figure 8.9 and parse table 8.10. We start with a prediction stack **`Session_1 #`**, a look-ahead **`!`** and a global counter which now stands at 2. For non-terminal **`Session`** and look-ahead **`!`** the table predicts **`Session -> Facts Question`**. So we generate the parse tree grammar rule **`Session_1 -> Facts_2 Question_3`** where **`Session_1`** obtains its number from the prediction and the **`Facts_2`** and **`Question_3`** obtain their numbers from the global counter. Next we turn the prediction stack into **`Facts_2 Question_3 #`**. For **`Facts`** and **`!`** the parse table yields the prediction **`Facts -> Fact Facts`** which gives us the parse tree grammar rule **`Facts_2 -> Fact_4 Facts_5`** and a stack **`Fact_4 Facts_5 Question_3 #`**. The next step is similar and produces the grammar rule **`Fact_4 -> ! STRING`** and a stack **`! STRING Facts_5 Question_3 #`**. Now we are ready to match the **`!`**.

This process generates successive layers of the parse tree, using non-terminal names like **`Question_3`** and **`Facts_5`** as forward pointers. See Figure 8.14, where the leaves of the tree spell the absorbed input followed by the prediction stack. When the parsing is finished, the leaves spell the input string.

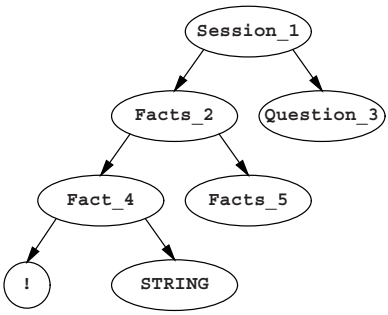


Fig. 8.14. Partial parse tree/grammar for input starting with **`! STRING ...`**

There is no need to clean the resulting grammar. It cannot have undefined non-terminals: each non-terminal created in a right-hand side is also put on the prediction stack and a subsequent prediction will create a rule for it. It cannot have unreachable

non-terminals either: rules are only created for non-terminals in the prediction stack, and these all derive from predictions that ultimately derive from the start symbol.

8.5 Extended LL(1) Grammars

Several parser generators accept an extended context-free grammar instead of an ordinary one. See for example Lewi et al. [46], Heckmann [49], and Grune and Jacobs [362]. Extended context-free grammars have been discussed in Chapter 2. To check that an extended context-free grammar is LL(1), we have to transform the extended context-free grammar into an ordinary one, in a way that will avoid introducing LL(1) conflicts. For example, the transformation for **`Something`**$^+$ given in Chapter 2:

`Something`$^+$ → **`Something`** | **`Something Something`**$^+$

will not do, because it will result in an LL(1) conflict on the symbols in FIRST(**`Something`**). Instead, we will use the following transformations:

`Something`* → ε | **`Something Something`***
`Something`$^+$ → **`Something Something`***
`Something`$^?$ → ε | **`Something`**

If the resulting grammar is LL(1), the original extended context-free grammar was ELL(1) (Extended LL(1)). This is the recursive interpretation of Chapter 2. Parser generation usually proceeds as follows: first transform the grammar to an ordinary context-free grammar, and then produce a parse table for it.

Extended LL(1) grammars allow a more efficient implementation in recursive descent parsers. In this case, **`Something`**$^?$ can be implemented as an **if** statement:

if look_ahead ∈ FIRST(**`Something`**) **then**
    code for **`Something`** ...
else if look_ahead ∉ FOLLOW(**`Something`**$^?$) **then**
    ERROR;

`Something`* can be implemented as a **while** loop:

while look_ahead ∈ FIRST(**`Something`**) **do**
    code for **`Something`** ...
if look_ahead ∉ FOLLOW(**`Something`***) **then**
    ERROR;

and **`Something`**$^+$ can be implemented as a **repeat** loop:

repeat
    if look_ahead ∉ FIRST(**`Something`**) **then**
        ERROR;
    code for **`Something`** ...
until look_ahead ∈ FOLLOW(**`Something`**$^+$);

Here procedure calls have been replaced by much more efficient repetitive constructs.

8.6 Conclusion

LL(1) parsing is a method with a strong intuitive appeal: the parser reads the input from left to right, making decisions on its next step based on its expectation (prediction stack) and the next token in the input. In some sense it "just follows the signs". The process can be implemented conveniently as a set of mutually recursive routines, one for each non-terminal. If there were no ε-rules, that would be about the whole story.

An ε-rule does not produce tokens, so it does not provide signs to follow. Instead it is transparent, which makes us consider the set of tokens that can occur after it. This set is dynamic, and it cannot be precomputed, but derives from the prediction. It can, however, be approximated from above, by using the FOLLOW set; a precomputable linear-time parser results.

The power of deterministic LL parsing can be increased by extending the look-ahead, to bounded length, resulting in LL(k) parsing, or to unbounded length, in LL-regular parsing. Linear-approximate LL(2) is a convenient and simplified form of LL(2) parsing.

Problems

Problem 8.1: Under what conditions is a grammar LL(0)? What can be said about the language it produces?

Problem 8.2: An LL(1) grammar is converted to CNF, as in Section 4.2.3. Is it still LL(1)?

Problem 8.3: In an LL(1) grammar all non-terminals that have only one alternative are substituted out. Is the resulting grammar still LL(1)?

Problem 8.4: What does it mean when a column for a token t in an LL(1) parse table is completely empty (for example **#** in Figure 8.5)?

Problem 8.5: *a*. Is the following grammar LL(1)?

```
S_S  →  A b | A c
 A   →  ε
```

Check with your local LL(1) parser generator. *b*. Same question for

```
S_S  →  A b | A c
 A   →  a | ε
```

c. Same question for

```
S_S  →  A
 A   →  a A
```

Problem 8.6: In Section 8.2.5.1 we give a simple argument showing that no left-recursive grammar can be LL(1): the union of the FIRST sets of the non-left-recursive alternatives would be equal the FIRST set of the left-recursive alternative, thus causing massive FIRST/FIRST conflicts. But what about the grammar

```
S_S  ->  S B | ε
B    ->  ε
```

which is left-recursive, obviously not LL(1), but the FIRST sets of both alternatives contain only ε?

Problem 8.7: Devise an algorithm to check if a given parse table could have originated from an LL(1) grammar through the LL(1) parse table construction process.

Problem 8.8: Devise an efficient table structure for an LL(k) parser where k is fairly large, say between 5 and 20. (Such grammars may arise in grammatical data compression, Section 17.5.1.)

9

Deterministic Bottom-Up Parsing

There is a great variety of deterministic bottom-up parsing methods. The first deterministic parsers (Wolpe [110], Adams and Schlesinger [109]) were bottom-up parsers and interest has only increased since. The full bibliography of this book on its web site contains about 280 entries on deterministic bottom-up parsing against some 85 on deterministic top-down parsing. These figures may not directly reflect the relative importance of the methods, but they are certainly indicative of the fascination and complexity of the subject of this chapter.

There are two families of deterministic bottom-up parsers:

- Pure bottom-up parsers. This family comprises the precedence and bounded-(right)-context techniques, and are treated in Sections 9.1 to 9.3.
- Bottom-up parsers with an additional top-down component. This family, which is both more powerful and more complicated than the pure bottom-up parsers, consists of the LR techniques and is treated in Sections 9.4 to 9.10.

There are two main ways in which deterministic bottom-up methods are extended to allow more grammars to be handled:

- Remaining non-determinism is resolved by breadth-first search. This leads to Generalized LR parsing, which is covered in Section 11.1.
- The requirement that the bottom-up parser does the reductions in reverse rightmost production order (see below and Section 3.4.3.2) is dropped. This leads to non-canonical parsing, which is covered in Chapter 10.

The proper setting for the subject at hand can best be obtained by summarizing a number of relevant facts from previous chapters.

- A rightmost production expands the rightmost non-terminal in a sentential form, by replacing it by one of its right-hand sides, as explained in Section 2.4.3. A sentence is then produced by repeated rightmost production until no non-terminal remains. See Figure 9.1(*a*), where the sentential forms are right-aligned to show how the production process creeps to the left, where it terminates. The grammar used is that of Figure 7.8.

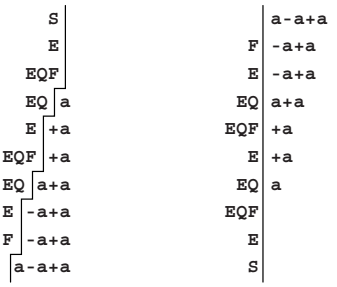


Fig. 9.1. Rightmost production (*a*) and rightmost reduction (*b*)

- Each step of a bottom-up parser, working on a sentential form, identifies the latest rightmost production in it and undoes it by reducing a segment of the input to the non-terminal it derived from. The identified segment and the production rule are called the "handle" (Section 3.4.3.2).

  Since the parser starts with the final sentential form of the production process (that is, the input) it finds its first reduction somewhere near to the left end, which is convenient for stream-based input. A bottom-up parser identifies rightmost productions in reverse order. See Figure 9.1(*b*) where the handles are left-aligned to show how the reduction process condenses the input.
- To obtain an efficient parser we need an efficient method to identify handles, without considering alternative choices. So the handle search must either yield *one* handle, in which case it must be the proper one, or *no* handle, in which case we have found an error in the input.

Although this chapter is called "Deterministic Bottom-Up *Parsing*", it is almost exclusively concerned with methods for finding handles. Once the handle is found, parsing is (almost always) trivial. The exceptions will be treated separately.

Unlike top-down parsing, which identifies productions before any of its constituents have been identified, bottom-up parsing identifies a production only at its very end, when all its constituents have already been identified. A top-down parser allows semantic actions to be performed at the beginning of a production and these actions can help in determining the semantics of the constituents. In a bottom-up parser, semantic actions are only performed during a reduction, which occurs at the end of a production, and the semantics of the constituents have to be determined without the benefit of knowing in which production they occur. We see that the increased power of bottom-up parsing compared to top-down parsing comes at a price: since the decision what production applies is postponed to the last moment, that decision can be based upon the fullest possible information, but it also means that the actions that depend on this decision come very late.

9.1 Simple Handle-Finding Techniques

There is a situation in daily life in which the average citizen is called upon to identify a handle. If one sees a formula like

$$4 + 5 \times 6 + 8$$

one immediately identifies the handle and evaluates it:

$$4 + \underline{5 \times 6} + 8$$

$$4 + \quad 30 \quad + 8$$

The next handle is

$$\underline{4 + \quad 30} \quad + 8$$

$$34 \qquad + 8$$

and then

$$\underline{34 \qquad + 8}$$

$$42$$

If we look closely, we can discern shifts and reduces in this process. People doing the arithmetic shift symbols until they reach the situation

$$4 + 5 \times 6 \qquad\qquad + 8$$

in which the control mechanism in their heads tells them that this is the right moment to do a reduce. If asked why, they might answer something like: "Ah, well, I was taught in school that multiplication comes before addition". Before we formalize this notion and turn it into a parsing method, we consider an even simpler case.

Meanwhile we note that formulas like the one above are called "arithmetic expressions" and are produced by the grammar of Figure 9.2. **S** is the start symbol, **E**

```
S_S  →  E
 E   →  E + T
 E   →  T
 T   →  T × F
 T   →  F
 F   →  n
 F   →  ( E )
```

Fig. 9.2. A grammar for simple arithmetic expressions

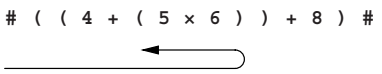
stands for "expression", **T** for "term", **F** for "factor" and **n** for any number. Having **n** rather than an explicit number causes no problems, since the exact value is immaterial to the parsing process. We have demarcated the beginning and the end of the

expression with # marks; the blank space that normally surrounds a formula is not good enough for automatic processing. The parser accepts the input as correct and stops when the input has been reduced to $\#S_S\#$.

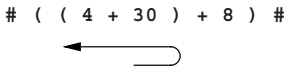
An arithmetic expression is *fully parenthesized* if each operator together with its operands has parentheses around it:

$$
\begin{array}{lcl}
S_S & \rightarrow & E \\
E & \rightarrow & (\; E \; + \; T \;) \\
E & \rightarrow & T \\
T & \rightarrow & (\; T \; \times \; F \;) \\
T & \rightarrow & F \\
F & \rightarrow & n
\end{array}
$$

Our example expression would have the form



Now finding the handle is trivial: go to the first closing parenthesis and then back to the nearest opening parenthesis. The segment between and including the parentheses is the handle and the operator identifies the production rule. Reduce it and repeat the process as often as required. Note that after the reduction there is no need to start all over again, looking for the first closing parenthesis: there cannot be any closing parenthesis on the left of the reduction spot. So we can start searching right where we are. In the above example we find the next right parenthesis immediately and do the next reduction:



9.2 Precedence Parsing

Of course, grammars normally do not have these convenient begin- and end markers to each compound right-hand side, and the above parsing method has little practical value (as far as we know it does not even have a name). Yet, suppose we had a method for inserting the proper parentheses into an expression that was lacking them. At a first glance this seems trivial to do: when we see **+n×** we know we can replace this by **+(n×** and we can replace **×n+** by **×n)+**. There is a slight problem with **+n+**, but since the first **+** has to be performed first, we replace this by **+n)+**. The **#**s are easy; we can replace **#n** by **#(n** and **n#** by **n)#**. For our example we get:

```
# ( 4 + ( 5 × 6 ) + 8 ) #
```

This is, however, not quite correct — it should have been **#((4+(5×6))+8)#** — and for **4+5×6** we get the obviously incorrect form **#(4+(5×6)#**.

9.2.1 Parenthesis Generators

The problem is that we do not know how many parentheses to insert in, for example, `+n×`: in `4+5×6` we should replace it by `+(n×` to obtain `#(4+(5×6))#`, but `4+5×6×76×8` would require it to be replaced by `+(((n×`, etc. We solve this problem by inserting *parenthesis generators* rather than parentheses. A generator for open parentheses is traditionally written as $\lessdot$, one for closing parentheses as $\gtrdot$; we shall also use a "non-parenthesis", $\doteq$. These symbols look confusingly like $<$, $>$ and $=$, to which they are only remotely related. Now our tentatively inserted parentheses become firmly inserted parenthesis generators; see Figure 9.3. We have left out the

$$
\begin{array}{ccccc}
+ & \times & \Rightarrow & + \lessdot & \times \\
\times & + & \Rightarrow & \times \gtrdot & + \\
+ & + & \Rightarrow & + \gtrdot & + \\
\# \cdots & & \Rightarrow & \# \lessdot & \cdots \\
\cdots & \# & \Rightarrow & \cdots \gtrdot & \#
\end{array}
$$

Fig. 9.3. Preliminary table of precedence relations

`n` since the parenthesis generator is dependent on the left and right operators only.

The table in Figure 9.3 is incomplete: the pattern `× ×` is missing, as are all patterns involving parentheses. In principle there should be a pattern for each combination of two operators (where we count the genuine parentheses as operators), and only the generator to be inserted is relevant for each combination. This generator is called the *precedence relation* between the two operators. It is convenient to collect all combinations of operators in a table, the *precedence table*. The precedence table for the grammar of Figure 9.2 is given in Figure 9.4; the leftmost column contains the left-hand symbols and the top-most row the right-hand symbols.

	#	+	×	(	)
#	$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$	
+	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
×	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(		$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$		$\gtrdot$

Fig. 9.4. Operator-precedence table to the grammar of Figure 9.2

There are three remarks to be made about this precedence table. First, we have added a number of $\lessdot$ and $\gtrdot$ tokens not covered above (for example, `×`$\gtrdot$`×`). Second, there is `#`$\doteq$`#` and `(`$\doteq$`)` — but there is no `)`$\doteq$`(` ! We shall shortly see what they mean. And third, there are three empty entries. When we find these combinations in the input, it contains an error.

Such a table is called a precedence table because for symbols that are normally regarded as operators it gives their relative precedence. An entry like `+`$\lessdot$`×` indicates

that in the combination `+×`, the `×` has a higher precedence than the `+`. We shall first show how the precedence table is used in parsing and then how such a precedence table can be constructed systematically for a given grammar, if the grammar allows it.

The stack in an operator-precedence parser differs from the normal bottom-up parser stack in that it contains "important" symbols, the operators, between which relations are defined, and "unimportant" symbols, the numbers, which are only consulted to determine the value of a handle and which do not influence the parsing. Moreover, we need locations on the stack to hold the parenthesis generators between the operators (although one could, in principle, do without these locations, by reevaluating the parenthesis generators again whenever necessary). Since there is a parenthesis generator between each pair of operators and there is also (almost) always a value between such a pair, we shall indicate both in the same position on the stack, with the parenthesis generator in line and the value below it; see Figure 9.5.

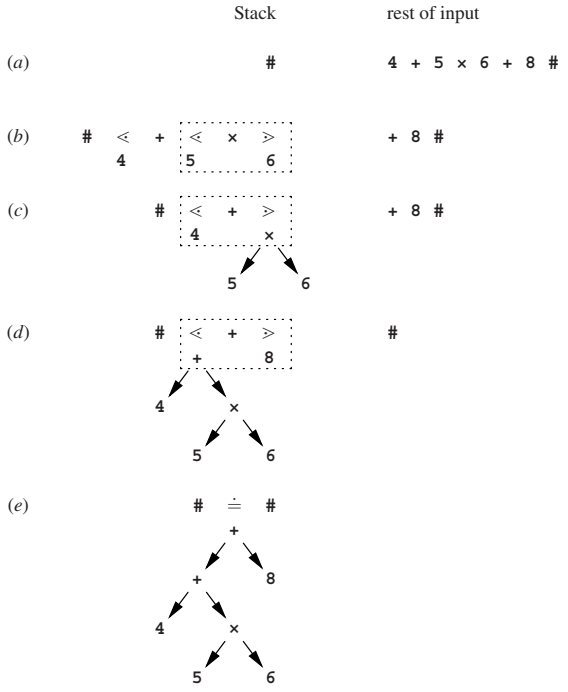


Fig. 9.5. Operator-precedence parsing of `4+5×6+8`

To show that, contrary to what is sometimes thought, operator-precedence can do more than just compute a value (and since we have seen too often now that `4+5×6+8=42`), we shall have the parser construct the parse tree rather than the value. The stack starts with a `#`. Values and operators are shifted onto it, interspersed with parenthesis generators, until a ⋗ generator is met; the following operator is not

shifted and is left in the input (Figure 9.5(*b*)). It is now easy to identify the handle segment, which is demarcated by a dotted rectangle in the figure. The operator `×` identifies the type of node to be created, and the handle is now reduced to a tree; see (*c*), in which also the next ⩾ has already appeared between the `+` on the stack and the `+` in the input. We see that the tree and the new generator have come in the position of the ⩽ of the handle. A further reduction brings us to (*d*) in which the `+` and the `8` have already been shifted, and then to the final state of the operator-precedence parser, in which the stack holds `#`≐`#` and the parse tree dangles from the value position.

We see that the stack only holds ⩽ markers and values, plus a ⩾ on the top each time a handle is found. The meaning of the ≐ becomes clearer when we parse an input text which includes parentheses, like `4×(5+6)`; see Figure 9.6. We see that the

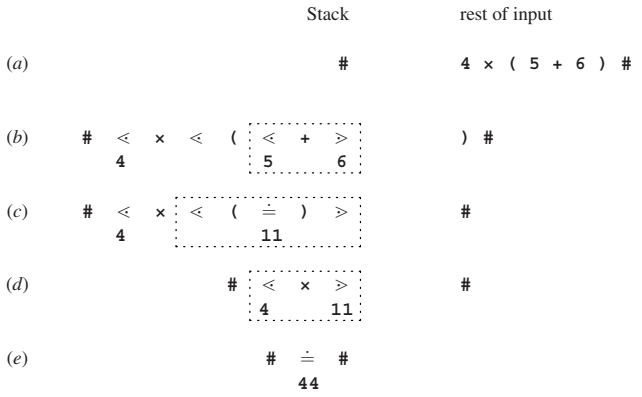


Fig. 9.6. An operator-precedence parsing involving ≐

≐ is used to build handles consisting of more than one operator and two operands; the handle in (*c*) has two operators, the `(` and the `)` and one operand, the `11`. Where the ⩽ generates open parentheses and the ⩾ generates close parentheses, both of which cause level differences in the parse tree, the ≐ generates no parentheses and allows the operands to exist on the same level in the parse tree.

As already indicated on page 200, the set of stack configurations of a bottom-up parser can be described by a regular expression. For precedence parsers the expression is easy to see:

`#` | `#`⩽`q` `(`[⩽≐]`q`)* ⩾? | `#`≐`#`

where `q` is any operator; the first alternative is the start situation and the third alternative is the end situation. (Section 9.12.2 will show more complicated regular expressions for other bottom-up parsers.)

9.2.2 Constructing the Operator-Precedence Table

The above hinges on the difference between operators, which are terminal symbols and between which precedence relations are defined, and operands, which are non-

terminals. This distinction is captured in the following definition of an operator grammar:

> A CF grammar is an *operator grammar* if (and only if) each right-hand side contains at least one terminal or non-terminal and no right-hand side contains two consecutive non-terminals.

So each pair of non-terminals is separated by at least one terminal; all the terminals except those carrying values (`n` in our case) are called operators.

For such grammars, setting up the precedence table is relatively easy. First we compute for each non-terminal A the set $FIRST_{OP}(A)$, which is the set of all operators that can occur as the first operator in sentential forms deriving from A. Note that such a first operator can be preceded by at most one non-terminal in an operator grammar. The FIRST_{OP}s of all non-terminals are constructed simultaneously as follows:

1. For each non-terminal A, find all right-hand sides of all rules for A; now for each right-hand side R we insert the first operator in R (if any) into $\text{FIRST}_{OP}(A)$. This gives us the initial values of all FIRST_{OP}s.
2. For each non-terminal A, find all right-hand sides of all rules for A; now for each right-hand side R that starts with a non-terminal, say B, we add the elements of $\text{FIRST}_{OP}(B)$ to $\text{FIRST}_{OP}(A)$. This is reasonable, since a sentential form of A may start with B, so all operators in $\text{FIRST}_{OP}(B)$ should also be in $\text{FIRST}_{OP}(A)$.
3. Repeat step 2 above until no FIRST_{OP} changes any more. We have now found the FIRST_{OP} of all non-terminals.

We will also need the set $LAST_{OP}(A)$, which is defined similarly, and a similar algorithm, using the *last* operator in R in step 1 and a B which *ends* A in step 2 provides it. The sets for the grammar of Figure 9.2 are shown in Figure 9.7.

$\text{FIRST}_{OP}(\texttt{S}) = \{\texttt{\#}\}$	$\text{LAST}_{OP}(\texttt{S}) = \{\texttt{\#}\}$
$\text{FIRST}_{OP}(\texttt{E}) = \{\texttt{+}, \texttt{×}, \texttt{(}\}$	$\text{LAST}_{OP}(\texttt{E}) = \{\texttt{+}, \texttt{×}, \texttt{)}\}$
$\text{FIRST}_{OP}(\texttt{T}) = \{\texttt{×}, \texttt{(}\}$	$\text{LAST}_{OP}(\texttt{T}) = \{\texttt{×}, \texttt{)}\}$
$\text{FIRST}_{OP}(\texttt{F}) = \{\texttt{(}\}$	$\text{LAST}_{OP}(\texttt{F}) = \{\texttt{)}\}$

Fig. 9.7. FIRST_{OP} and LAST_{OP} sets for the grammar of Figure 9.2

Now we can fill the precedence table using the following rules, in which q, q_1 and q_2 are operators and A is a non-terminal.

- For each occurrence in a right-hand side of the form $q_1\ q_2$ or $q_1\ A\ q_2$, set $q_1 \doteq q_2$. This keeps operators from the same handle together.
- For each occurrence $q_1\ A$, set $q_1 \lessdot q_2$ for each q_2 in $\text{FIRST}_{OP}(A)$. This demarcates the left end of a handle.
- For each occurrence Aq_1, set $q_2 \gtrdot q_1$ for each q_2 in $\text{LAST}_{OP}(A)$. This demarcates the right end of a handle.

If we obtain a table without conflicts this way, that is, if we never find two different relations between two operators, then we call the grammar *operator-precedence*.

It will now be clear why `(`$\doteq$`)` and not `)`$\doteq$`(` in our grammar of Figure 9.2, and why `+`$\gtrdot$`+`: because `E+` occurs in `E`$\rightarrow$`E+T` and `+` is in LAST_{OP}(`E`).

In this way, the table can be derived from the grammar by a program and be passed on to the operator-precedence parser. A very efficient linear-time parser results. There is, however, one small problem we have glossed over: Although the method properly identifies the handle segment, it often does not identify the non-terminal to which to reduce it. Also, it does not show any unit rule reductions; nowhere in the examples did we see reductions of the form `E`$\rightarrow$`T` or `T`$\rightarrow$`F`. In short, operator-precedence parsing generates only *skeleton parse trees*.

Operator-precedence parsers are very easy to construct (often even by hand) and very efficient to use; operator-precedence is the method of choice for all parsing problems that are simple enough to allow it. That only a skeleton parse tree is obtained, is often not an obstacle, since operator grammars often have the property that the semantics is attached to the operators rather than to the right-hand sides; the operators are identified correctly.

It is surprising how many grammars are (almost) operator-precedence. Almost all formula-like computer input is operator-precedence. Also, large parts of the grammars of many computer languages are operator-precedence. An example is a construction like `CONST total = head + tail;` from a Pascal-like language, which is easily rendered as:

									Stack	rest of input
`#`	$\lessdot$	`CONST`	$\lessdot$	`total`	`=`	$\lessdot$	`head`	`+`	$\gtrdot$	`; #`
									`tail`	

Ignoring the non-terminals has other bad consequences besides producing a skeleton parse tree. Since non-terminals are ignored, a missing non-terminal is not noticed. As a result, the parser will accept incorrect input without warning and will produce an incomplete parse tree for it. A parser using the table of Figure 9.4 will blithely accept the empty string, since it immediately leads to the stack configuration `#`$\doteq$`#`. It produces a parse tree consisting of one empty node.

The theoretical analysis of this phenomenon turns out to be inordinately difficult; see Levy [125], Williams [128, 129, 131] and many others in (Web)Section 18.1.6. In practice it is less of a problem than one would expect; it is easy to check for the presence of required non-terminals, either while the parse tree is being constructed or afterwards — but such a check would not follow from the parsing technique.

9.2.3 Precedence Functions

Although precedence tables require room for only a modest $|V_T|^2$ entries, where $|V_T|$ is the number of terminals in the grammar, they can often be represented much more frugally by so-called *precedence functions*, and it is usual to do so. The idea is the following. Rather than having a *table* T such that for any two operators q_1 and q_2, $T[q_1, q_2]$ yields the relation between q_1 and q_2, we have two integer *functions* f and g such that $f(q_1) < g(q_2)$ means that $q_1 \lessdot q_2$, $f(q_1) = g(q_2)$ means $q_1 \doteq q_2$

and $f(q_1) > g(q_2)$ means $q_1 \gtrdot q_2$. $f(q)$ is called the *left priority* of q, $g(q)$ the *right priority*; they would probably be better indicated by l and r, but the use of f and g is traditional. It will be clear that *two* functions are required: with just one function one cannot express, for example, `+`$\gtrdot$`+`. Precedence functions take much less room than precedence tables: $2|V_T|$ entries versus $|V_T|^2$ for the table. Not all tables allow a representation with two precedence functions, but many do.

Finding the proper f and g for a given table seems simple enough and can indeed often be done by hand. The fact, however, that there are two functions rather than one, the size of the tables and the occurrence of the $\doteq$ complicate things. An algorithm to construct the two functions was given by Bell [120]. There is always a way to represent a precedence table with more than two functions; Bertsch [127] shows how to construct such functions.

Finding two precedence functions is equivalent to reordering the rows and columns of the precedence table so that the latter can be divided into three regions: a $\gtrdot$ region on the lower left, a $\lessdot$ region on the upper right and a $\doteq$ border between them; see Figure 9.8. The process is similar but not equivalent to doing a topological

	#	)	+	×	(
#	$\doteq$		$\lessdot$	$\lessdot$	$\lessdot$
(		$\doteq$	$\lessdot$	$\lessdot$	$\lessdot$
+	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$
×	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	

Fig. 9.8. The precedence table of Figure 9.4 reordered

sort on f_q and g_q.

Precedence parsing recognizes that many languages have tokens that define the structure and tokens that carry the information; the first are the operators, the second the operands. That raises the question whether that difference can be formalized; see Gray and Harrison [124] for a partial answer, but usually the question is left to the user.

Some operators are actually composite; the C and Java programming language conditional expression, which is formed by two parts: `x>0?x:0` yields `x` if `x` is greater than 0; otherwise it yields 0. Such distributed operators are called *distfix operators*. They can be handled by precedence-like techniques; see, for example Peyton Jones [132] and Aasa [133].

9.2.4 Further Precedence Methods

Operator precedence structures the input in terms of operators only: it yields skeleton parse trees — correctly structured trees with the terminals as leaves but with unlabeled nodes — rather than parse trees. As such it is quite powerful, and serves in many useful programs to this day. In some sense it is even stronger than the more

famous LR techniques: operator precedence can easily handle ambiguous grammars, as long as the ambiguity remains restricted to the labeling of the tree. We could add a rule **E→n** to the grammar of Figure 9.2 and it would be ambiguous but still operator-precedence. It achieves its partial superiority over LR by not fulfilling the complete task of parsing: getting a completely labeled parse tree.

There is a series of more advanced precedence parsers, which do properly label the parse tree with non-terminals. They were very useful at the time they were invented, but today their usefulness has been eclipsed by the LALR and LR parsers, which we will treat further on in this chapter (Sections 9.4 through 9.14). We will therefore only briefly touch upon them here, and refer the reader to the many publications in (Web)Section 18.1.6.

The most direct way to bring back the non-terminals in the parse tree is to involve them like the terminals in the precedence relations. This idea leads to *simple precedence* parsing (Wirth and Weber [118]). A grammar is simple precedence if and only if:

- it has a conflict-free precedence table over all its symbols, terminals and non-terminals alike;
- none of its right-hand sides is ε;
- all of its right-hand sides are different.

For example, we immediately have the precedence relations **(**$\doteq$**E** and **E**$\doteq$**)** from the rule **F→(E)**.

The construction of the simple-precedence table is again based upon two sets, $FIRST_{\mathrm{ALL}}(A)$ and $LAST_{\mathrm{ALL}}(A)$. $\mathrm{FIRST}_{\mathrm{ALL}}(A)$ is similar to the set FIRST(A) from Section 8.2.1.1, and differs from it in that it also contains all non-terminals that can start a sentential form derived from A, whereas FIRST(A) contains terminals only. A similar definition applies to $\mathrm{LAST}_{\mathrm{ALL}}(A)$.

Unfortunately almost no grammar is simple-precedence, not even the simple grammar of Figure 9.2, since we have **(**$\lessdot$**E** in addition to **(**$\doteq$**E**, due to the occurrence of **(E** in **F→(E)**, and **E** being in $\mathrm{FIRST}_{\mathrm{ALL}}$(**E**) from **E→E+T**. A few other conflicts also occur. On the bright side, this kind of conflict can often be solved by inserting extra levels around the troublesome non-terminals, as done in Figure 9.9, but this brings us farther away from our goal, producing a correct parse tree.

It turns out that most of the simple-precedence conflicts are $\lessdot/\doteq$ conflicts. Now the difference between $\lessdot$ and $\doteq$ is in a sense less important than that between either of them and $\gtrdot$. Both $\lessdot$ and $\doteq$ result in a shift and only $\gtrdot$ asks for a reduce. Only when a reduce is found will the difference between $\lessdot$ and $\doteq$ become significant for finding the left end of the handle. Now suppose we drop the difference between $\lessdot$ and $\doteq$ and combine them into $\leqslant$; then we need a different means of identifying the handle segment. This can be done by requiring not only that all right-hand sides be different, but also that no right-hand side be equal to the tail of another right-hand side. A grammar that conforms to this and has a conflict-free $\leqslant/\gtrdot$ precedence table is called *weak precedence* (Ichbiah and Morse [121]).

```
S_S  →  E'
E'   →  E
 E   →  E + T'
 E   →  T'
T'   →  T
 T   →  T × F
 T   →  F
 F   →  n
 F   →  ( E )
```

$\text{FIRST}_{\text{ALL}}(\texttt{E'}) = \{\texttt{E}, \texttt{T'}, \texttt{T}, \texttt{F}, \texttt{n}, \texttt{(}\}$ $\quad \text{LAST}_{\text{ALL}}(\texttt{E'}) = \{\texttt{T'}, \texttt{T}, \texttt{F}, \texttt{n}, \texttt{)}\}$
$\text{FIRST}_{\text{ALL}}(\texttt{E}) = \{\texttt{E}, \texttt{T'}, \texttt{T}, \texttt{F}, \texttt{n}, \texttt{(}\}$ $\quad \text{LAST}_{\text{ALL}}(\texttt{E}) = \{\texttt{T}, \texttt{F}, \texttt{n}, \texttt{)}\}$
$\text{FIRST}_{\text{ALL}}(\texttt{T'}) = \{\texttt{T}, \texttt{F}, \texttt{n}, \texttt{(}\}$ $\quad \text{LAST}_{\text{ALL}}(\texttt{T'}) = \{\texttt{F}, \texttt{n}, \texttt{)}\}$
$\text{FIRST}_{\text{ALL}}(\texttt{T}) = \{\texttt{T}, \texttt{F}, \texttt{n}, \texttt{(}\}$ $\quad \text{LAST}_{\text{ALL}}(\texttt{T}) = \{\texttt{F}, \texttt{n}, \texttt{)}\}$
$\text{FIRST}_{\text{ALL}}(\texttt{F}) = \{\texttt{n}, \texttt{(}\}$ $\quad \text{LAST}_{\text{ALL}}(\texttt{F}) = \{\texttt{n}, \texttt{)}\}$

	#	E'	E	T'	T	F	n	+	×	(	)
#		≐	⋖	⋖	⋖	⋖	⋖			⋖	
E'	≐										
E	⋗							≐			≐
T'	⋗							⋗			⋗
T	⋗							⋗	≐		⋗
F	⋗							⋗	⋗		⋗
n	⋗							⋗	⋗		⋗
+				≐	⋖	⋖	⋖			⋖	
×						≐	⋖			⋖	
(			≐	⋖	⋖	⋖	⋖			⋖	
)	⋗							⋗	⋗		⋗

Fig. 9.9. Modifying the grammar from Figure 9.2, to obtain a conflict-free simple-precedence table

Unfortunately the simple grammar of Figure 9.2 is not weak-precedence either. The right-hand side of **E→T** is the tail of the right-hand side of **E→E+T**, and upon finding the stack

$$\cdots \lessdot \texttt{E} \doteq \texttt{+} \lessdot \texttt{T} \gtrdot$$

we do not know whether to reduce with **E→T** or with **E→E+T**. Several tricks are possible: taking the longest reduce, looking deeper on the stack, etc.

The above methods determine the precedence relations by looking at 1 symbol on the stack and 1 token in the input. Once this has been said, the idea suggests itself to generalize this and to determine the precedence relations from the topmost *m* symbols on the stack and the first *n* tokens in the input. This is called *(m,n)-extended precedence* (Wirth and Weber [118]). For many entries in the table checking the full length on the stack and in the input is overkill, and ways have been found to use just

enough information, thus greatly reducing the table sizes. This technique is called *mixed-strategy precedence* (McKeeman [123]).

9.3 Bounded-Right-Context Parsing

There is a different way to solve the annoying problem of the identification of the right-hand side: let the identity of the rule be part of the precedence relation. This means that for each combination of, say, m symbols on the stack and n tokens in the input there should be a unique parsing decision which is either "shift" ($\lessdot$) or "reduce using rule X" ($\gtrdot_X$), as obtained by a variant of the rules for extended precedence. The parser is then a form of *bounded-right-context*. Figure 9.10 gives such tables for $m = 2$ and $n = 1$ for the grammar of Figure 9.2; these tables were constructed by hand. The rows correspond to stack symbol pairs; the entry Accept means that

	#	+	×	n	(	)
#S	Accept					
#E	$\gtrdot_{S\to E}$	$\lessdot$				Error
#T	$\gtrdot_{E\to T}$	$\gtrdot_{E\to T}$	$\lessdot$			Error
#F	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$			Error
#n	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	Error
#(	Error	Error	Error	$\lessdot$	$\lessdot$	Error
E+	Error	Error	Error	$\lessdot$	$\lessdot$	Error
E)	$\gtrdot_{F\to(E)}$	$\gtrdot_{F\to(E)}$	$\gtrdot_{F\to(E)}$	Error	Error	$\gtrdot_{F\to(E)}$
T×	Error	Error	Error	$\lessdot$	$\lessdot$	Error
+T	$\gtrdot_{E\to E+T}$	$\gtrdot_{E\to E+T}$	$\lessdot$			$\gtrdot_{E\to E+T}$
+F	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$			$\gtrdot_{T\to F}$
+n	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	$\gtrdot_{F\to n}$
+(	Error	Error	Error	$\lessdot$	$\lessdot$	Error
×F	$\gtrdot_{T\to T\times F}$	$\gtrdot_{T\to T\times F}$	$\gtrdot_{T\to T\times F}$			$\gtrdot_{T\to T\times F}$
×n	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	$\gtrdot_{F\to n}$
×(	Error	Error	Error	$\lessdot$	$\lessdot$	Error
(E	Error	$\lessdot$				$\lessdot$
(T	Error	$\gtrdot_{E\to T}$	$\lessdot$			$\gtrdot_{E\to T}$
(F	Error	$\gtrdot_{T\to F}$	$\gtrdot_{T\to F}$			$\gtrdot_{T\to F}$
(n	Error	$\gtrdot_{F\to n}$	$\gtrdot_{F\to n}$	Error	Error	$\gtrdot_{F\to n}$
((	Error	Error	Error	$\lessdot$	$\lessdot$	Error

Fig. 9.10. BC(2,1) table for the grammar of Figure 9.2

the input has been parsed and Error means that a syntax error has been found. Blank entries will never be accessed; all-blank rows have been left out. See, for example, Loeckx [122] for an algorithm for the construction of such tables.

9.3.1 Bounded-Context Techniques

The table of Figure 9.10 represents a variant of bounded-context, or more precisely, a particular implementation of bounded-right-context. To understand the bounded-context idea we have to go back to the basic bottom-up parsing algorithm explained in Section 3.2.2: find a right-hand side anywhere in the sentential form and reduce it. But we have already seen that often such a reduction creates a node that is not a node of the final parse tree, and backtracking is needed. For example, when we reduce the second **n** in **#n×n#** to **F** we have created a node that belongs to the parse tree. We obtain **#n×F#**, but if we now reduce the **F** to **T**, obtaining **#n×T#**, we have gone one step too far, and will no longer get a parsing. So why is the first reduction OK, and the second is not?

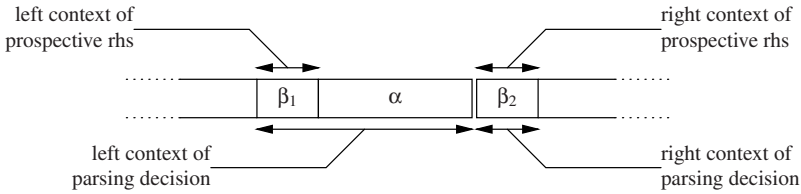
In bounded-context parsing the proposed reductions are restricted by context conditions. A right-hand side α of a rule $A \rightarrow \alpha$ found in a sentential form can only be reduced to A if it appears in the right context, $\beta_1\alpha\beta_2$. Here β_1 is the left context, β_2 the right one. Both contexts must be of bounded length, hence "bounded context"; either or both can be ε.

Using these contexts, it is easy to see from the grammar that **n** in the context **×···#** can be reduced to **F**, but **F** in the context **×···#** cannot be reduced to **T**, although in the context **+···#** it could. Turning this intuition into an algorithm is very difficult. A grammar is *bounded-context* if no segment $\beta_1\alpha\beta_2$ that results from a production $A \rightarrow \alpha$ in a sentential form can result in any other way. If that condition holds, we can, upon seeing the context pattern $\beta_1\alpha\beta_2$, safely reduce to $\beta_1 A\beta_2$. If the maximum length of β_1 is m and that of β_2 is n, the grammar is *BC(m,n)*.

Finding sufficient and non-conflicting contexts is a difficult affair, which is sketched by Floyd [117]. Because of this difficulty, bounded-context is of no consequence as a parsing method; but bounded-context grammars are important in error recovery (Richter [313], Ruckert [324]) and substring parsing (Cormack [211], Ruckert [217]), since they allow parsing to be resumed in arbitrary positions. This property is treated in Section 16.5.2.

If all right contexts in a bounded-context grammar contain terminals only, the grammar and its parser are *bounded-right-context*, or *BRC(m,n)*. Much more is known about bounded-right-context than about general bounded-context, and extensive table construction algorithms are given by Eickel et al. [115] and Loeckx [122]. Table construction is marginally easier for BRC than for BC, but it can handle fewer grammars.

The implementation of BRC parsing as sketched above is awkward: to try a reduction $A \rightarrow \alpha$ in the context $\beta_1 \cdots \beta_2$ the top of the stack must be tested for the presence of α, which is of variable length, and then β_1 on the stack and β_2 in the input must be verified; repeat for all rules and all contexts. It is much more convenient to represent all triplets $(\beta_1\alpha\beta_2)$ as pairs $(\beta_1\alpha, \beta_2)$ in a matrix, like the one in Figure 9.10; in this way $\beta_1\alpha$ and β_2 are basically the left and right contexts of the parsing decision at the gap between stack and rest of input:



As a final step the left contexts are cut to equal lengths, in such a way that enough information remains. This is usually easily done; see Figure 9.10. This brings BRC parsing in line with the other table-driven bottom-up parsing algorithms.

Although some publications do not allow it, BC and BRC parsers can handle nullable non-terminals. If we add the rule **E→**ε to the grammar of Figure 9.2, the context **(,)** is strong enough to conclude that reducing with **E→**ε is correct.

Bounded-right-context is much more prominent than bounded-context, but since it is more difficult to pronounce, it is often just called "bounded-context"; this sometimes leads to considerable confusion. BRC(2,1) is quite powerful and was once very popular, usually under the name "BC(2,1)", but has been superseded almost completely by LALR(1) (Section 9.7).

It should be pointed out that bounded-context can identify reductions in non-canonical order, since a context reduction may be applied anywhere in the sentential form. Such a reduction can then result in a non-terminal which is part of the right context of another reduction pattern. So bounded context actually belongs in Chapter 10, but is easier to understand here.

If in bounded-*right*-context we repeatedly apply the first context reduction we find in a left-to-right sweep, we identify the reductions in canonical order, since the right context is free from non-terminals all the time, so no non-canonical reductions are needed.

If during table construction for a bounded-context parser we find that a segment $\beta_1\alpha\beta_2$ produced from $\beta_1 A\beta_2$ can also be produced otherwise, we can do two things: we can decide that the grammar is not BC and give up, or we can decide not to include the segment in our table of reduction contexts and continue. In doing so we now run the risk of losing some parsings, unless we can prove that for any sentential form there is at least one reduction context left. If that is the case, the grammar is *bounded-context parsable* or *BCP*. Constructing parse tables for BCP(*m*,*n*) is even more difficult than for BC or BRC, but the method can handle substantially more grammars than either; Williams [193] has the details.

Note that the parsing method is the same for BRC, BC and BCP; just the parse table construction methods differ.

9.3.2 Floyd Productions

Bounded-context parsing steps can be summarized conveniently by using Floyd productions. *Floyd productions* are rules for rewriting a string that contains a marker, Δ, on which the rules focus. A Floyd production has the form $\alpha\Delta\beta \Rightarrow \gamma\Delta\delta$ and means

that if the marker in the string is preceded by α and is followed by β, the construction must be replaced by $\gamma\Delta\delta$. The rules are tried in order starting from the top and the first one to match is applied; processing then resumes on the resulting string, starting from the top of the list, and the process is repeated until no rule matches.

Although Floyd productions were not primarily designed as a parsing tool but rather as a general string manipulation language, the identification of the Δ in the string with the gap in a bottom-up parser suggests itself and was already made in Floyd's original article [113]. Floyd productions for the grammar of Figure 9.2 are given in Figure 9.11. The parser is started with the Δ at the left of the input.

```
Δ n      =>  n Δ
Δ (      =>  ( Δ
n Δ      =>  F Δ
T Δ ×    =>  T× Δ
T×F Δ    =>  T Δ
F Δ      =>  T Δ
E+T Δ    =>  E Δ
T Δ      =>  E Δ
(E) Δ    =>  F Δ
Δ +      =>  + Δ
Δ )      =>  ) Δ
Δ #      =>  # Δ
#E# Δ    =>  S Δ
```

Fig. 9.11. Floyd productions for the grammar of Figure 9.2

The apparent convenience and conciseness of Floyd productions makes it very tempting to write parsers in them by hand, but Floyd productions are very sensitive to the order in which the rules are listed and a small inaccuracy in the order can have a devastating effect.

9.4 LR Methods

The LR methods are based on the combination of two ideas that have already been touched upon in previous sections. To reiterate, the problem is to find the handle in a sentential form as efficiently as possible, for as large a class of grammars as possible. Such a handle is searched for from left to right. Now, from Section 5.10 we recall that a very efficient way to find a string in a left-to-right search is by constructing a finite-state automaton. Just doing this is, however, not good enough. It is quite easy to construct an FS automaton that would recognize any of the right-hand sides in the grammar efficiently, but it would just find the leftmost reducible substring in the sentential form. This substring, however, often does not identify the correct handle.

The idea can be made practical by applying the same trick that was used in the Earley parser to drastically reduce the fan-out of the breadth-first search (see Section 7.2): start the automaton with the start rule of the grammar and only consider,

in any position, right-hand sides that could be derived from the start symbol. This top-down restriction device served in the Earley parser to reduce the cost to $O(n^3)$, here we require the grammar to be such that it reduces the cost to $O(n)$. The resulting automaton is started in its initial state at the left end of the sentential form and allowed to run to the right. It has the property that it stops at the right end of the handle segment and that its accepting state tells us how to reduce the handle; if it ends in an error state the sentential form was incorrect. Note that this accepting state is an accepting state of the handle-finding automaton, not of the LR parser; the latter accepts the input only when it has been completely reduced to the start symbol.

Once we have found the handle, we follow the standard procedure for bottom-up parsers: we reduce the handle to its parent non-terminal as described at the beginning of Chapter 7. This gives us a new "improved" sentential form, which, in principle should be scanned anew by the automaton from the left, to find the next handle. But since nothing has changed in the sentential form between its left end and the point of reduction, the automaton will go through the same movements as before, and we can save it the trouble by remembering its states and storing them between the tokens on the stack. This leads us to the standard setup for an LR parser, shown in Figure 9.12 (compare Figure 7.1). Here $\mathtt{s}_1$ is the initial state, $\mathtt{s}_g \cdots \mathtt{s}_b$ are the states from

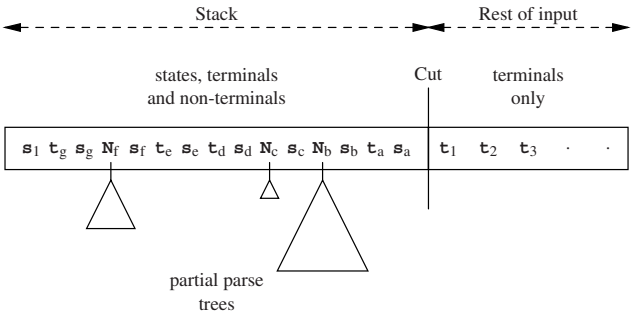


Fig. 9.12. The structure of an LR parse

previous scans, and $\mathtt{s}_a$ is the top, deciding, state.

By far the most important component in an LR parser is the handle-finding automaton, and there are many methods to construct one. The most basic one is LR(0) (Section 9.5); the most powerful one is LR(1) (Section 9.6); and the most practical one is LALR(1) (Section 9.7). In its decision process the LR automaton makes a very modest use of the rest of the input (none at all for LR(0) and a one-token look-ahead for LR(1) and LALR(1)); several extensions of LR parsing exist that involve the rest of the input to a much larger extent (Sections 9.13.2 and 10.2).

Deterministic handle-finding automata can be constructed for any CF grammar, which sounds promising, but the problem is that an accepting state may allow the automaton to continue searching in addition to identifying a handle (in which case we have a shift/reduce conflict), or identify more than one handle (and we have a reduce/reduce conflict). (Both types of conflicts are explained in Section 9.5.3.) In

other words, the automaton is deterministic; the attached semantics is not. If that happens the LR method used is not strong enough for the grammar. It is easy to see that there are grammars for which no LR method will be strong enough; the grammar of Figure 9.13 produces strings consisting of an odd number of **a**s, the middle of which is the handle. But finding the middle of a string is not a feature of

```
Ss  -> a S a | a
```

Fig. 9.13. An unambiguous non-deterministic grammar

LR parsers, not even of the extended and improved versions.

As with the Earley parser, LR parsers can be improved by using look-ahead, and almost all of them are. An LR parser with a look-ahead of k tokens is called LR(k). Just as the Earley parser, it requires k end-of-input markers to be appended to the input; this implies that an LR(0) parser does not need end-of-input markers.

9.5 LR(0)

Since practical handle-finding FS automata easily get so big that their states cannot be displayed on a single page of a book, we shall use the grammar of Figure 9.14 for our examples. It describes very simple arithmetic expressions, terminated with a **$**.

```
1.  Ss  ->  E $
2.   E  ->  E - T
3.   E  ->  T
4.   T  ->  n
5.   T  ->  ( E )
```

Fig. 9.14. A very simple grammar for differences of numbers

An example of a string in the language is **n-(n-n)$**; the **n** stands for any number. The only arithmetic operator in the grammar is the **-**; it serves to remind us that the proper parse tree must be derived, since **(n-n)-n$** is not the same as **n-(n-n)$**.

9.5.1 The LR(0) Automaton

We set out to construct a top-down-restricted handle-recognizing FS automaton for the grammar of Figure 9.14, and start by constructing a non-deterministic version. We recall that a non-deterministic automaton can be drawn as a set of states connected by arrows (transitions), each marked with one symbol or with ε. Each state will contain one *item*. Like in the Earley parser an item consists of a grammar rule with a dot • embedded in its right-hand side. An item $X \rightarrow \cdots Y \bullet Z \cdots$ in a state

means that the NFA bets on $X \rightarrow \cdots YZ \cdots$ being the handle and that it has already recognized $\cdots Y$. Unlike the Earley parser there are no back-pointers.

To simplify the explanation of the transitions involved, we introduce a second kind of state, which we call a *station*. It has only ε-arrows incoming and outgoing, contains something of the form $\bullet X$ and is drawn in a rectangle rather than in an ellipse. When the automaton is in such a station at some point in the sentential form, it assumes that at this point a handle starts which reduces to X. Consequently each $\bullet X$ station has ε-transitions to items for all rules for X, each with the dot at the left end, since no part of the rule has yet been recognized; see Figure 9.15. Equally reasonably, each state holding an item $X \rightarrow \cdots \bullet Z \cdots$ has an ε-transition to the station $\bullet Z$, since the bet on an X may be over-optimistic and the automaton may have to settle for a Z. The third and last source of arrows in the NFA is straightforward. From each state containing $X \rightarrow \cdots \bullet P \cdots$ there is a P-transition to the state containing $X \rightarrow \cdots P \bullet \cdots$, for P a terminal or a non-terminal. This corresponds to the move the automaton makes when it really meets a P. Note that the sentential form may contain non-terminals, so transitions on non-terminals should also be defined.

With this knowledge we refer to Figure 9.15. The stations for **S**, **E** and **T** are

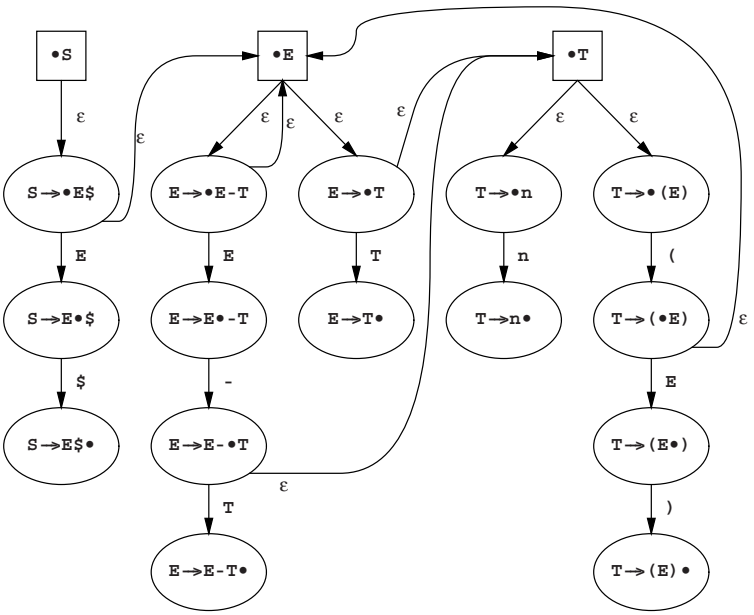


Fig. 9.15. A non-deterministic handle recognizer for the grammar of Figure 9.14

drawn at the top of the picture, to show how they lead to all possible items for **S**, **E** and **T**, respectively. From each station ε-arrows fan out to all states containing items with the dot at the left, one for each rule for the non-terminal in that station; from each such state non-ε-arrows lead down to further states. Now the picture is almost

complete. All that needs to be done is to scan the items for a dot followed by a non-terminal (readily discernible from the outgoing arrow marked with it) and to connect each such item to the corresponding station through an ε-arrow. This completes the picture.

There are three things to be noted about this picture. First, for each grammar rule with a right-hand side of length l there are $l+1$ items and they are easily found in the picture. Moreover, for a grammar with r different non-terminals, there are r stations. So the number of states is roughly proportional to the size of the grammar, which assures us that the automaton will have a modest number of states. For the average grammar of a hundred rules something like 300 states is usual. The second thing to note is that all states have outgoing arrows except the ones which contain a reduce item, an item with the dot at the right end. These are accepting states of the automaton and indicate that a handle has been found; the item in the state tells us how to reduce the handle. The third thing to note about Figure 9.15 is its similarity to the recursive transition network representation of Section 2.8.

We shall now run this NFA on the sentential form **E-n-n$**, to see how it works. As in the FS case we can do so if we are willing to go through the trouble of resolving the non-determinism on the fly. The automaton starts at the station **•S** and can immediately make ε-moves to **S→•E$**, **•E**, **E→•E-T**, **E→•T**, **•T**, **T→•n** and **T→•(E)**. Moving over the **E** reduces the set of items to **S→E•$** and **E→E•-T**; moving over the next **-** brings us at **E→E-•T** from which ε-moves lead to **•T**, **T→•n** and **T→•(E)**. Now the move over **n** leaves only one item: **T→n•**. Since this is a reduce item, we have found a handle segment, **n**, and we should reduce it to **T** using **T→n**. See Figure 9.16. This reduction gives us a new sentential form, **E-T-n$**, on which we can repeat the process.

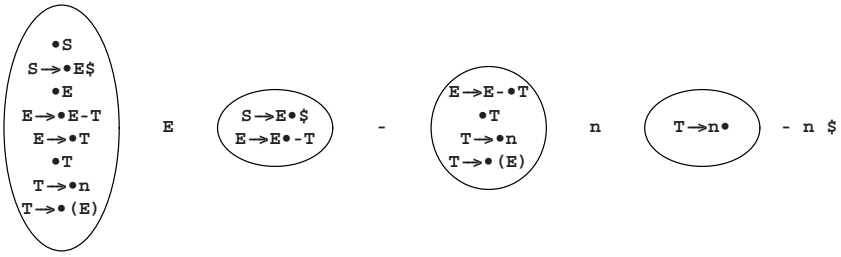


Fig. 9.16. The sets of NFA states while analysing **E-n-n$**

We see that there are two ways in which new items are produced: through ε-moves and through moving over a symbol. The first way yields items of the form $A \rightarrow \bullet\alpha$, and such an item derives from an item of the form $X \rightarrow \beta\bullet A\gamma$ in the same state. The second way yields items of the form $A \rightarrow \alpha\sigma\bullet\beta$ where σ is the token we moved over; such an item derives from an item of the form $A \rightarrow \alpha\bullet\sigma\beta$ in the parent state.

Just as in the FS case (page 144) we have described an interpreter for the nondeterministic handle recognizer, and the first thing we need to do is to make the NFA deterministic, if we are to use this parsing method in earnest. We use the subset construction of Section 5.3.1 to construct a deterministic automaton with the sets of the items of Figure 9.15 as its states. The result is shown in Figure 9.17, where we have

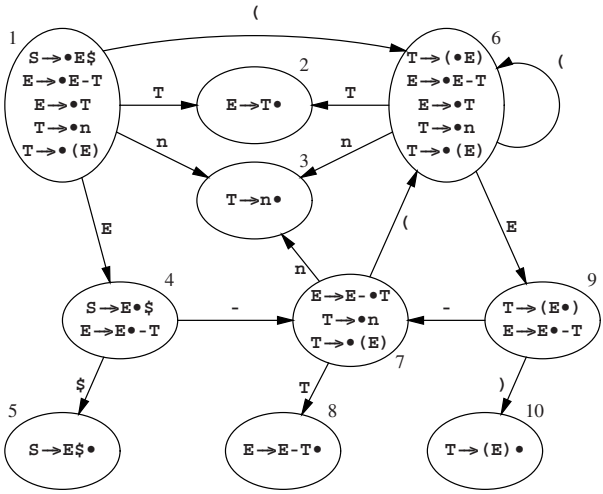


Fig. 9.17. The corresponding deterministic handle recognizer

left out the stations to avoid clutter and because they are evident from the other items. We see that the deterministic automaton looks a lot less understandable than Figure 9.15; this is the price one has to pay for having determinism. Yet we see that the subset construction has correctly identified the subsets we had already constructed by hand in Figure 9.16. This type of automaton is called an *LR(0) automaton*.

9.5.2 Using the LR(0) Automaton

It is customary to number the states of the deterministic automaton, as has already been done in Figure 9.17 (the order of the numbers is arbitrary; they serve identification purposes only). Now it has become much easier to represent the sentential form with its state information, both in a program and in a drawing:

① **E** ④ **-** ⑦ **n** ③ **- n $**

The sequence ① ④ ⑦ ③ can be read from Figure 9.17 using the path **E-n**. We start with state ① on the stack and shift in symbols from the sentential form, all the while assessing the new states. As soon as an accepting state shows up on the top of the stack (and it cannot show up elsewhere on the stack) the shifting stops and a reduce is called for; the accepting state indicates how to reduce. Accepting state ③ calls for a reduction **T→n**, so our new sentential form will be **E-T-n$**.

Repeating the handle-finding process on this new form we obtain the configuration

```
① E ④ - ⑦ T ⑧                    - n $
```

which shows us two things. First, the automaton has landed in state ⑧ and thus identified a new reduction, **E→E-T**, which is correct. Second, we see here the effect already hinted at in Figure 9.12: by restarting the automaton at the beginning of the sentential form we have done superfluous work. Up to state ⑦, that is, up to the left end of the handle **T→n**, nothing had changed, so we could have saved work if we had remembered the old states ①, ④, and ⑦ between the symbols in the sentential form.

This leads to the following LR(0) parsing algorithm:

1. Consult the state s on top of the stack; it is either an accepting state specifying a reduction $X \to \alpha$ or it is a non-accepting state.
   a) If s is an accepting state, unstack $|\alpha|$ pairs of symbols and states from the stack, where $|\alpha|$ is the length of the right-hand side α. The unstacked symbols constitute the children in the parse tree for X; see Figure 9.12. Next we push X onto the stack. We have now reduced α to X.
   b) If s is a non-accepting state, shift the next token from the input onto the stack.
2. The top of the stack is now a non-terminal (1a.) or terminal (1b.) symbol T, with a state u under it. Find state u in the LR(0) automaton and follow the path marked T starting from that state.
   a) If this leads to a state v, push v onto the stack.
   b) Otherwise the input is erroneous.

Two things are important about this algorithm. The first is that if we start with a consistent stack configuration (each triple of state, symbol, and state on the stack corresponds to a transition in the LR(0) automaton) the stack configuration will again be consistent afterwards. And the second is that it does the proper reductions, and thus does the parsing we were looking for.

Note that the state u exposed after a reduction can never call for another reduction: if it did, that reduction would already have been performed earlier.

We see that LR(0) parsing is performed in two steps: 1. the top state indicates an action, shift or reduce with a given rule, which is then performed; 2. a new top state is computed by going from one state through a transition to another state. It is convenient to represent an LR(0) automaton in an *ACTION table* and a *GOTO table*, both indexed by states. The GOTO table has columns indexed by symbols; the ACTION table has just one column. In step 1 we consult the ACTION table based on the state; in step 2 we index the GOTO table with a given symbol and a given state to find the new state. The LR(0) ACTION and GOTO tables for the automaton of Figure 9.17 are given in Figure 9.18.

Suppose we find state 6 on top of the stack and the next input token is **n**. The ACTION table tells us to shift, and then the GOTO table, at the intersection of 6 and **n**, tells us to stack the state 3. And the ACTION table for state 3 tells us to reduce

	ACTION
1	shift
2	`E → T`
3	`T → n`
4	shift
5	`S → E $`
6	shift
7	shift
8	`E → E - T`
9	shift
10	`T → ( E )`

	GOTO						
	`n`	`-`	`(`	`)`	`$`	`E`	`T`
1	3	e	6	e	e	4	2
2							
3							
4	e	7	e	e	5		
5							
6	3	e	6	e	e	9	2
7	3	e	6	e	e		8
8							
9	e	7	e	10	e		
10							

Fig. 9.18. LR(0) ACTION and GOTO tables for the grammar of Figure 9.14

using `T→n`. An entry "e" means that an error has been found: the corresponding symbol cannot legally appear in that position. A blank entry will never even be consulted: either the state calls for a reduction or the corresponding symbol will never at all appear in that position, regardless of the form of the input. In state 4, for example, we will never meet an `E`: the `E` would have originated from a previous reduction, but no reduction would do that in that position. Since non-terminals are only put on the stack in legal places no empty entry on a non-terminal will ever be consulted.

In practice the ACTION entries for reductions do not directly refer to the rules to be used, but to the numbers of these rules. These numbers are then used to index an array of routines that have built-in knowledge of the rules, that know how many entries to unstack and that perform the semantic actions associated with the recognition of the rule in question. Parts of these routines will be generated by a parser generator. Also, the reduce and shift information is combined in one table, the *ACTION/GOTO table*, with entries of the forms "s*N*", "r*N*" or "e". An entry "s*N*" means "shift the input symbol onto the stack and go to state *N*", which is often abbreviated to "shift to *N*". An entry "r*N*" means "reduce by rule number *N*"; the shift over the resulting non-terminal has to be performed afterwards. And "e" means error, as above. The ACTION/GOTO table for the automaton of Figure 9.17 is given in Figure 9.19.

Tables like in Figures 9.18 and 9.19 contain much empty space and are also quite repetitious. As grammars get bigger, the parsing tables get larger and they contain progressively more empty space and redundancy. Both can be exploited by data compression techniques and it is not uncommon that a table can be reduced to 15% of its original size by the appropriate compression technique. See, for example, Al-Hussaini and Stone [67] and Dencker, Dürre and Heuft [338].

The advantages of LR(0) over precedence and bounded-right-context are clear. Unlike precedence, LR(0) immediately identifies the rule to be used for reduction, and unlike bounded-right-context, LR(0) bases its conclusions on the entire left context rather than on the last *m* symbols of it. In fact, LR(0) can be seen as a clever implementation of BRC(∞,0), i.e., bounded-right-context with unrestricted left context and zero right context.

	n	-	(	)	$	E	T
1	s3	e	s6	e	e	s4	s2
2	r3	r3	r3	r3	r3	r3	r3
3	r4	r4	r4	r4	r4	r4	r4
4	e	s7	e	e	s5		
5	r1	r1	r1	r1	r1	r1	r1
6	s3	e	s6	e	e	s9	s2
7	s3	e	s6	e	e		s8
8	r2	r2	r2	r2	r2	r2	r2
9	e	s7	e	s10	e		
10	r5	r5	r5	r5	r5	r5	r5

Fig. 9.19. The ACTION/GOTO table for the grammar of Figure 9.14

9.5.3 LR(0) Conflicts

By now the reader may have the vague impression that something is wrong. On the one hand we claim that there is no known method to make a linear-time parser for an arbitrary grammar; on the other we have demonstrated above a method that seems to work for an arbitrary grammar. An NFA as in Figure 9.15 can certainly be constructed for any grammar, and the subset construction will certainly turn it into a deterministic one, which will definitely not require more than linear time. Voilà, a linear-time parser.

The problem lies in the accepting states of the deterministic automaton. An accepting state may still have an outgoing arrow, say on a symbol **+**, and if the next symbol is indeed a **+**, the state calls for both a reduction and for a shift: the combination of automaton and interpretation of the accepting states is not really deterministic after all. Or an accepting state may be an honest accepting state but call for two different reductions. The first problem is called a *shift/reduce conflict* and the second a *reduce/reduce conflict*. Figure 9.20 shows examples (which derive from a slightly different grammar than in Figure 9.14).

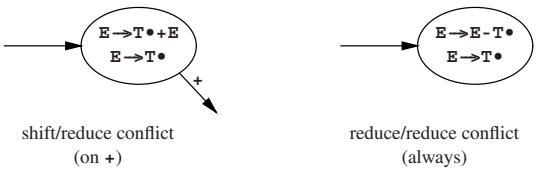


Fig. 9.20. Two types of conflict

Note that there cannot be a shift/shift conflict. A shift/shift conflict would imply that two different arrows leaving the same state would carry the same symbol. This is, however, prevented by the subset algorithm (which would have made into one the two states the arrows point to).

A state that contains a conflict is called an *inadequate state*. A grammar that leads to a deterministic LR(0) automaton with no inadequate states is called *LR(0)*. The absence of inadequate states in Figure 9.17 proves that the grammar of Figure 9.14 is LR(0).

9.5.4 ε-LR(0) Parsing

Many grammars would be LR(0) if they did not have ε-rules. The reason is that a grammar with a rule $A \rightarrow \varepsilon$ cannot be LR(0): from any station $P \rightarrow \cdots \bullet A \cdots$ an ε-arrow leads to a state $A \rightarrow \bullet$ in the non-deterministic automaton, which causes a DFA state containing both the shift item $P \rightarrow \cdots \bullet A \cdots$ and the reduce item $A \rightarrow \bullet$. And this state is inadequate, since it exhibits a shift/reduce conflict. We shall now look at a partial solution to this obstacle: ε-LR(0) parsing.

The idea is to do the ε-reductions required by the reduce part of the shift/reduce conflict already while constructing the DFA. Normally reduces cannot be precomputed since they require the first few top elements of the parsing stack, but obviously that problem does not exist for ε-reductions.

The grammar of Figure 9.21, a variant of the one in Figure 7.17, contains an ε-rule and hence is not LR(0). (The ε-rule is intended to represent multiplication.) The

```
S_S  →  E $
 E   →  E Q F
 E   →  F
 F   →  a
 Q   →  /
 Q   →  ε
```

Fig. 9.21. An ε-LR(0) grammar

start item **S→•E$** leads to **E→•EQF** by an ε-move, and from there to **E→E•QF** by a move over **E**. This item has two ε-moves, to **Q→•/** and to **Q→•**; the second causes a shift/reduce conflict. Following the above plan, we apply the offending rule to the item **E→E•QF**, but the resulting item cannot be **E→EQ•F**, for two reasons. First, the same item would result from finding a **/** in the input; and second, there is no corresponding **Q** on the parsing stack. So we mark the **Q** in the new item with a stroke on top: $\bar{\mathtt{Q}}$, to indicate that it does not correspond to a **Q** on the parse stack, a kind of non-**Q**.

We can now remove the item **Q→•** since it has played its part; the shift/reduce conflict is gone, and a deterministic handle recognizer results. This means that the grammar is ε-*LR(0)*; the deterministic handle recognizer is shown in Figure 9.22. The endangered state is state 4; the state that would result in "normal" LR(0) parsing is also shown, marked 4✘. We see that the immediate reduction **Q→**ε and the subsequent shift over **Q** have resulted in an item **F→•a** that is not present in the pure LR(0) state 4✘.

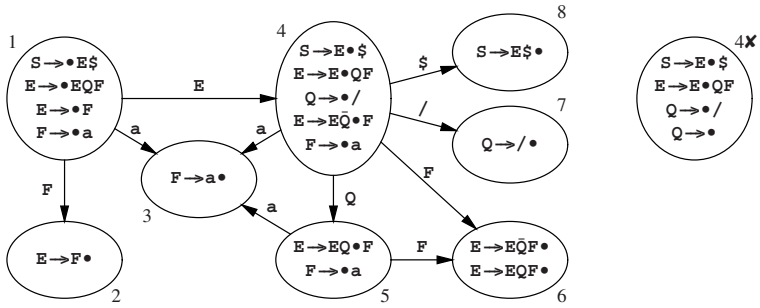


Fig. 9.22. Deterministic ε-LR(0) automaton for the grammar of Figure 9.21

In addition to the ε-reductions during parser table construction, ε-LR(0) parsing has another feature: when constructing the states of the deterministic handle recognizer, items that differ only in the presence or absence of bars over non-terminals are considered equal. So while the transition over **F** from state 4 yields an item **E→EQ̄F•** and that from state 5 yields **E→EQF•**, both transitions lead to state 6, which contains both items.

This feature has two advantages and one problem. The first advantage is that with this feature more grammars are ε-LR(0) than without it, although this plays no role in our example. The second is that the semantics of a single rule, the **E→EQF** in our example, is not split up over several items.

The problem is of course that we now have a reduce/reduce conflict. This problem is solved dynamically — during parsing — by checking the parse stack. If it contains

① **E** ④ **Q** ⑤ **F** ⑥ ...

we know the **Q** was there; we unstack 6 elements, perform the semantics of **E→EQF**, and push an **E**. If the parse stack contains

① **E** ④ **F** ⑥ ...

we know the **Q** was not there; we unstack 2 elements, create a node for **Q→**ε, unstack 2 more elements, perform the semantics of **E→EQF**, and push an **E**. Note that this modifies the basic behavior of the LR automaton, and it could thus be argued that ε-LR(0) parsing actually is not an LR technique.

Besides allowing grammars to be handled that would otherwise require much more complicated methods, ε-LR(0) parsing has the property that the non-terminals on the stack all correspond to non-empty segments of the input. This is obviously good for efficiency, but also very important in some more advanced parsing methods, for example generalized LR parsing (Section 11.1.4).

For more details on ε-LR(0) parsing and the related subject of hidden left recursion see Nederhof [156, Chapter 4], and Nederhof and Sarbo [94]. These also supply examples of grammars for which combining items with different bar properties is beneficial.

9.5.5 Practical LR Parse Table Construction

Above we explained the construction of the deterministic LR automaton (for example Figure 9.17) as an application of the subset algorithm to the non-deterministic LR automaton (Figure 9.15), but most LR parser generators (and many textbooks and papers) follow more closely the process indicated in Figure 9.16. This process combines the creation of the non-deterministic automaton with the subset algorithm: each step of the algorithm creates a transition $u \xrightarrow{t} v$, where u is an existing state and v is a new or old state. For example, the first step in Figure 9.16 created the transition ① $\xrightarrow{\mathbf{E}}$ ④. In addition the algorithm must do some bookkeeping to catch duplicate states. The LR(0) version works as follows; other LR parse table construction algorithms differ only in details.

The algorithm maintains a data structure representing the deterministic LR handle recognizer. Several implementations are possible, for example a graph like the one in Figure 9.17. Here we will assume it to consist of a list of pairs of states (item sets) and numbers, called S, and a set of transitions T. S represents the bubbles in the graph, with their contents and numbers; T represents the arrows. The algorithm also maintains a list U of numbers of new, unprocessed LR states. Since there is a one-to-one correspondence between states and state numbers we will use them interchangeably.

The algorithm starts off by creating a station $\bullet A$, where A is the start symbol of the grammar. This station is expanded, the resulting items are wrapped into a state numbered 1, the state is inserted into S, and its number is inserted in U. An item or a station I is expanded as follows:

1. If the dot is in front of a non-terminal A in I, create items of the form $A \rightarrow \bullet \cdots$ for all grammar rules $A \rightarrow \cdots$; then expand these items recursively until no more new items are created. The result of expanding I is the resulting item set; note that this is a set, so there are no duplicates. (This implements the ε-transitions in the non-deterministic LR automaton.)
2. If the dot is not in front of a non-terminal in I, the result of expanding I is just I.

The LR automaton construction algorithm repeatedly removes a state u from the list U and processes it by performing the following actions on it for all symbols (terminals and non-terminals) t in the grammar:

1. An empty item set v is created.
2. The algorithm finds items of the form $A \rightarrow \alpha \bullet t \beta$ in u. For each such item a new item $A \rightarrow \alpha t \bullet \beta$ is created, the kernel items. (This implements the vertical transitions in the non-deterministic LR automaton.) The created items are expanded as described above and the resulting items are inserted in v.
3. If state v is not already present in S, it is new and the algorithm adds it to U. Then v is added to S and the transition $u \xrightarrow{t} v$ is added to T. Here u was already present in S; the transition is certainly new to T; and v may or may not be new to S. Note that v may be empty; it is then the error state.

Since the above algorithm constructs all transitions, even those to error states, it builds a complete automaton (page 152).

The algorithm terminates because the work to be done is extracted from the list U, but only states not processed before are inserted in U. Since there are only a finite number of states, there must come a moment that there are no new states any more, after which the list U will become empty. And since the algorithm only creates states that are reachable and since only a very small fraction of all states are reachable, that moment usually arrives very soon.

9.6 LR(1)

Our initial enthusiasm about the clever and efficient LR(0) parsing technique will soon be damped considerably when we find out that very few grammars are in fact LR(0). If we drop the `$` from rule 1 in the grammar of Figure 9.14 since it does not really belong in arithmetic expressions, we find that the grammar is no longer LR(0). The new grammar is given in Figure 9.23, the non-deterministic automaton in Figure 9.24, and the deterministic one in Figure 9.25. State 5 has disappeared, since it was reached by a transition on `$`, but we have left the state numbering intact to facilitate comparison; a parser generator would of course number the states consecutively.

```
1.  S  →  E
2.  E  →  E - T
3.  E  →  T
4.  T  →  n
5.  T  →  ( E )
```

Fig. 9.23. A non-LR(0) grammar for differences of numbers

When we inspect the new LR(0) automaton, we observe to our dismay that state 4 (marked ✘) is now inadequate, exhibiting a shift/reduce conflict on `-`, and the grammar is not LR(0). This is all the more vexing as this is a rather stupid inadequacy: `S→E•` can never occur in front of a `-` but only in front of a `#`, the end-of-input marker, so there is no real problem at all. If we had developed the parser by hand, we could easily test in state 4 if the symbol ahead was a `-` or a `#` and act accordingly (or else there was an error in the input). Since, however, practical parsers have hundreds of states, such manual intervention is not acceptable and we have to find algorithmic ways to look at the symbol ahead.

Taking our cue from the explanation of the Earley parser,[1] we attach to each dotted item a look-ahead symbol. We shall separate the look-ahead symbol from the item by a space rather than enclose it between `[]`s as we did before, to avoid visual

[1] Actually LR parsing was invented (Knuth [52, 1965]) before Earley parsing (Earley [14, 1970]).

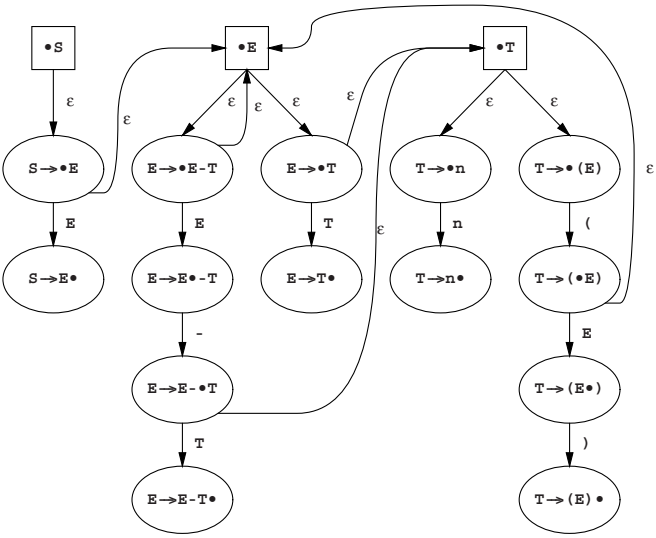


Fig. 9.24. NFA for the grammar in Figure 9.23

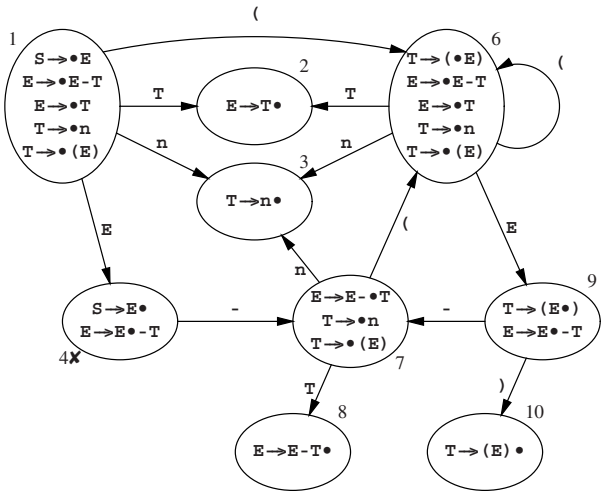


Fig. 9.25. Inadequate LR(0) automaton for the grammar in Figure 9.23

clutter. The construction of a non-deterministic handle-finding automaton using this kind of item, and the subsequent subset construction yield an LR(1) parser.

We shall now examine Figure 9.26, the NFA. Like the items, the stations have to carry a look-ahead symbol too. Actually, a look-ahead symbol in a station is more natural than that in an item: a station like •**E** **#** just means hoping to see an **E** followed by a **#**. The parser starts at station •**S** **#**, which has the end marker **#** as its look-ahead. From it we have ε-moves to all production rules for **S**, of which

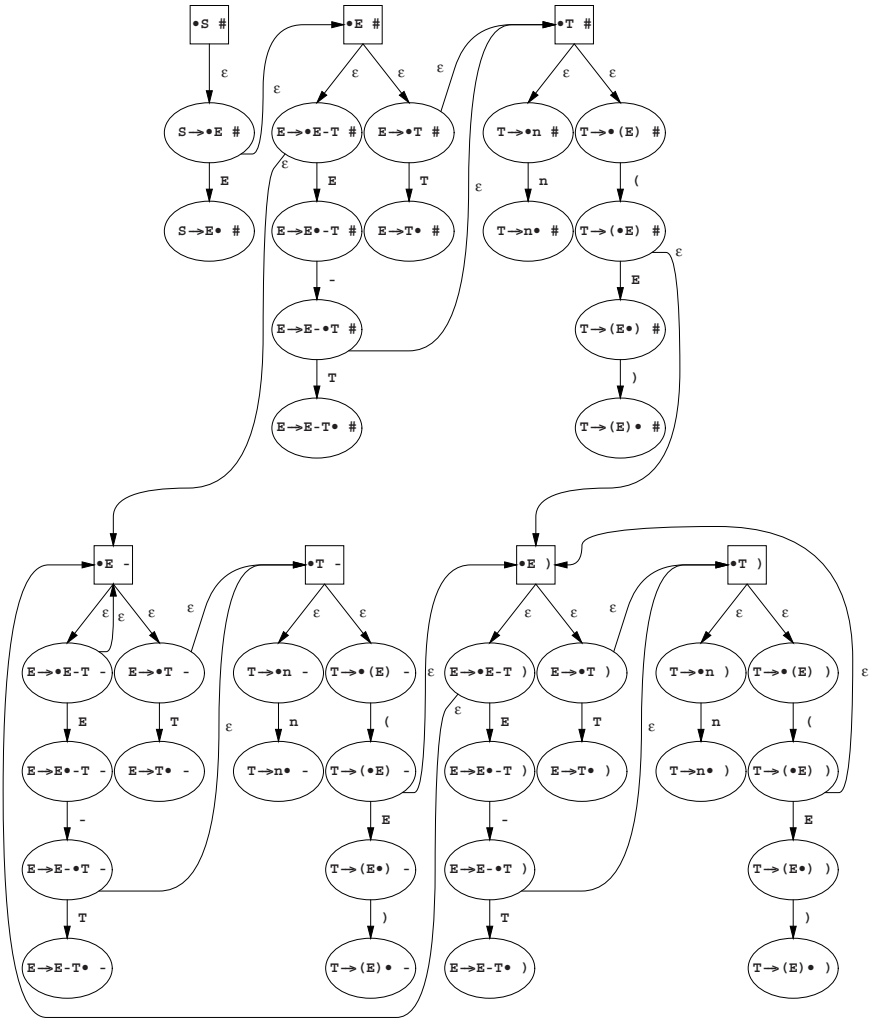


Fig. 9.26. Non-deterministic LR(1) automaton for the grammar in Figure 9.23

there is only one; this yields the item **S→•E #**. This item necessitates the station **•E #**; note that we do not automatically construct all possible stations as we did for the LR(0) automaton, but only those to which there are actual moves from elsewhere in the automaton. The station **•E #** produces two items by ε-transitions, **E→•E-T #** and **E→•E #**. It is easy to see how the look-ahead propagates. The item **E→•E-T #** in turn necessitates the station **•E -**, since now the automaton can be in the state "hoping to find an **E** followed by a **-**". The rest of the automaton will hold no surprises.

Look-aheads of items are directly copied from the items or stations they derive from; Figure 9.26 holds many examples. The look-ahead of a station derives either from the symbol following the originating non-terminal:

the item **E→•E-T** leads to station **•E -**

or from the previous look-ahead if the originating non-terminal is the last symbol in the item:

the item **S→•E #** leads to station **•E #**

There is a complication which does not occur in our example. When a non-terminal is followed by another non-terminal:

$P \rightarrow \bullet QR$

there will be ε-moves from this item to all stations $\bullet Q\ y$, where for y we have to fill in all terminals in FIRST(R). This is reasonable since all these and only these symbols can follow Q in this particular item. It will be clear that this is a rich source of stations. More complications arise when the grammar contains ε-rules, for example when R can produce ε; these are treated in Section 9.6.1.

The next step is to run the subset algorithm of page 145 on this automaton to obtain the deterministic automaton; if the automaton has no inadequate states, the grammar was *LR(1)* and we have obtained an LR(1) parser. The result is given in Figure 9.27. As was to be expected, it contains many more states than the LR(0)

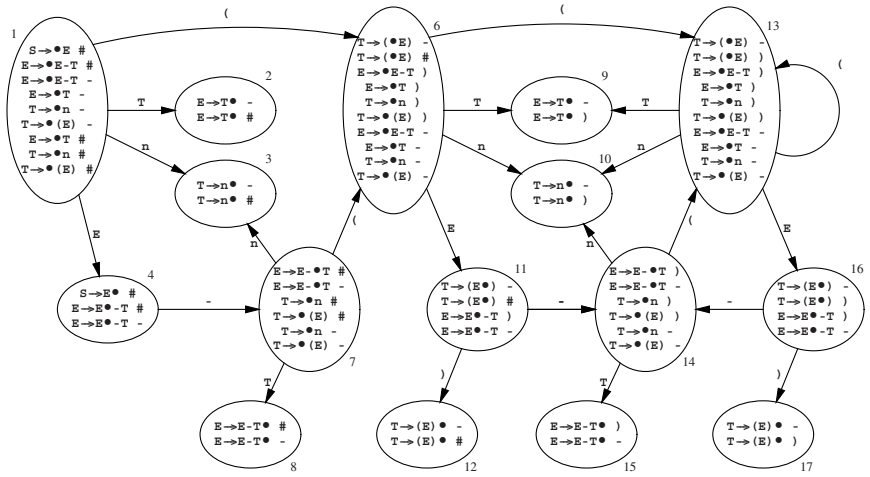


Fig. 9.27. Deterministic LR(1) automaton for the grammar in Figure 9.23

automaton although the 60% increase is very modest, due to the simplicity of the grammar. An increase of a factor of 10 or more is more likely in practice. (Although Figure 9.27 was constructed by hand, LR automata are normally created by a parser generator exclusively.)

We are glad but not really surprised to see that the problem of state 4 in Figure 9.25 has been resolved in Figure 9.27: on **#** reduce using **S→E**, on **-** shift to state 7 and on any other symbol give an error message.

It is again useful to represent the LR(1) automaton in an ACTION and a GOTO table; they are shown in Figure 9.28 (state 5 is missing, as explained on page 290). The combined ACTION/GOTO table can be obtained by superimposing both tables; this results in the LR(1) parsing table as it is used in practice.

ACTION

	n	**-**	**(**	**)**	**#**
1	s	e	s	e	e
2	e	r3	e	e	r3
3	e	r4	e	e	r4
4	e	s	e	e	r1
6	s	e	s	e	e
7	s	e	s	e	e
8	e	r2	e	e	r2
9	e	r3	e	r3	e
10	e	r4	e	r4	e
11	e	s	e	s	e
12	e	r5	e	e	r5
13	s	e	s	e	e
14	s	e	s	e	e
15	e	r2	e	r2	e
16	e	s	e	s	e
17	e	r5	e	r5	e

GOTO

	n	**-**	**(**	**)**	**#**	**S**	**E**	**T**
1	3		6			accept	4	2
2								
3								
4		7						
6	10		13				11	9
7	3		6					8
8								
9								
10								
11		14		12				
12								
13	10		13				16	9
14	10		13					15
15								
16		14		17				
17								

Fig. 9.28. LR(1) ACTION and GOTO tables for the grammar of Figure 9.23

The sentential form **E-n-n#** leads to the following configuration:

① **E** ④ **-** ⑦ **n** ③ **- n #**

and since the look-ahead is **-**, the correct reduction **T→n** is indicated.

All stages of the LR(1) parsing of the string **n-n-n** are given in Figure 9.29. Note that state ④ in *h* causes a shift (look-ahead **-**) while in *l* it causes a reduce (look-ahead **#**).

When we compare the ACTION and GOTO tables in Figures 9.28 and 9.18, we find two striking differences. First, the ACTION table now has several columns and is indexed with the look-ahead token in addition to the state; this is as expected. What is less expected is that, second, all the error entries have moved to the ACTION table. The reason is simple. Since the look-ahead was taken into account when constructing the ACTION table, that table orders a shift only when the shift can indeed be performed, and the GOTO step of the LR parsing algorithm does not need to do checks any more: the blank entries in the GOTO table will never be accessed.

a	①	n-n-n#	shift
b	① n ③	-n-n#	reduce 4
c	① T ②	-n-n#	reduce 3
d	① E ④	-n-n#	shift
e	① E ④ - ⑦	n-n#	shift
f	① E ④ - ⑦ n ③	-n#	reduce 4
g	① E ④ - ⑦ T ⑧	-n#	reduce 2
h	① E ④	-n#	shift
i	① E ④ - ⑦	n#	shift
j	① E ④ - ⑦ n ③	#	reduce 4
k	① E ④ - ⑦ T ⑧	#	reduce 2
l	① E ④	#	reduce 1
m	① S	#	accept

Fig. 9.29. LR(1) parsing of the string **n-n-n**

It is instructive to see how the LR(0) and LR(1) parsers react to incorrect input, for example **E-nn**$\cdots$. The LR(1) parser of Figure 9.28 finds the error as soon as the second **n** appears as a look-ahead:

① E ④ - ⑦ n ③ n$\cdots$

since the pair (3,**n**) in the ACTION table yields "e"; the GOTO table is not even consulted. The LR(0) parser of Figure 9.18 behaves differently. After reading **E-n** it is in the configuration

① E ④ - ⑦ n ③ n$\cdots$

where entry 3 in the ACTION table tells it to reduce by **T**$\rightarrow$**n**:

① E ④ - ⑦ T ⑧ n$\cdots$

and now entry 8 in the ACTION table tells it to reduce again, by **E**$\rightarrow$**E-T** this time:

① E ④ n$\cdots$

Only now is the error found, since the pair (4,**n**) in the GOTO table in Figure 9.18 yields "e".

Since the LR(0) automaton has fewer states than the LR(1) automaton, it retains less information about the input to the left of the handle; since it does not use look-ahead it uses less information about the input to the right of the handle. So it is not surprising that the LR(0) automaton is less alert than the LR(1) automaton.

9.6.1 LR(1) with ε-Rules

In Section 3.2.2 we have seen that one has to be careful with ε-rules in bottom-up parsers: they are hard to recognize bottom-up. Fortunately LR(1) parsers are strong enough to handle them without problems. In the NFA, an ε-rule is nothing special; it is just an exceptionally short list of moves starting from a station (see station •**Bc** in Figure 9.31(*a*)). In the deterministic automaton, the ε-reduction is possible in all

states of which the ε-rule is a member, but hopefully its look-ahead sets it apart from all other rules in those states. Otherwise a shift/reduce or reduce/reduce conflict results, and indeed the presence of ε-rules in a grammar raises the risks of such conflicts and reduces the likelihood of the grammar being LR(1).

```
S  →  A B c
A  →  a
B  →  b
B  →  ε
```

Fig. 9.30. A simple grammar with an ε-rule

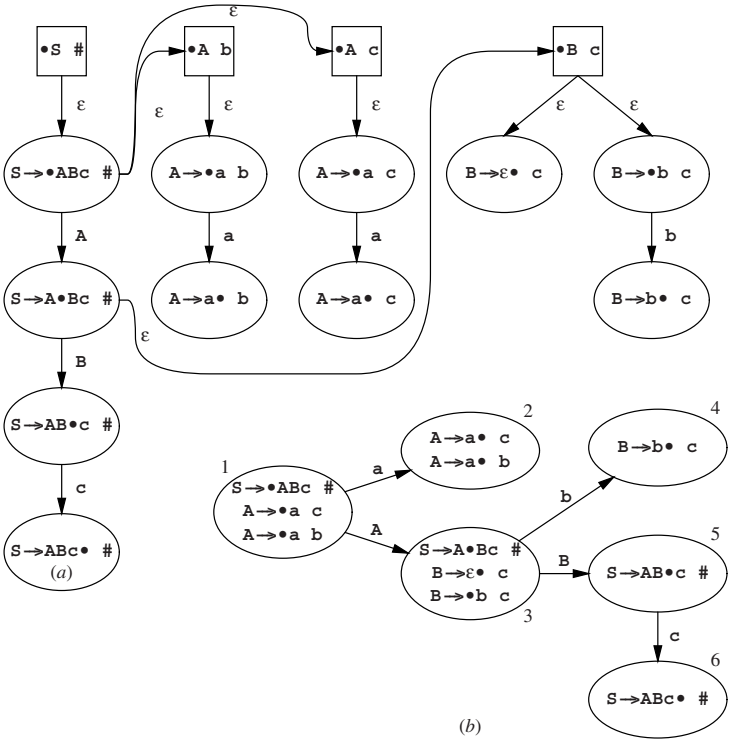


Fig. 9.31. Non-deterministic and deterministic LR(1) automata for Figure 9.30

To avoid page-filling drawings, we demonstrate the effect using the trivial grammar of Figure 9.30. Figure 9.31(*a*) shows the non-deterministic automaton, Figure 9.31(*b*) the resulting deterministic one. Note that no special actions were necessary to handle the rule **B**→ε.

The only complication occurs again in determining the look-ahead sets in rules in which a non-terminal is followed by another non-terminal; here we meet the same phenomenon as in an LL(1) parser (Section 8.2.2.1). Given an item, for example, $P \rightarrow \bullet ABC\ d$ where d is the look-ahead, we are required to produce the look-ahead set for the station $\bullet A\ \cdots$. If B had been a terminal, it would have been the look-ahead. Now we take the FIRST set of B, and if B produces ε (is nullable) we add the FIRST set of C since B can be transparent and allow us to see the first token of C. If C is also nullable, we may even see d, so in that case we also add d to the look-ahead set. The result of these operations can be written as FIRST(BCd). The new look-ahead set cannot turn out to be empty: the sequence of symbols from which it is derived (the BCd above) always ends in the original look-ahead set, and that was not empty.

9.6.2 LR($k > 1$) Parsing

Instead of a one-token look-ahead k tokens can be used, with $k > 1$. Surprisingly, this is not a straightforward extension of LR(1). The reason is that for $k > 1$ we also need to compute look-ahead sets for shift items. That this is so can be seen from the LR(2) grammar of Figure 9.32. It is clear that the grammar is not LR(1): the input must start

```
1. S_S  -> Aa | Bb | Cec | Ded
2.  A   -> qE
3.  B   -> qE
4.  C   -> q
5.  D   -> q
6.  E   -> e
```

Fig. 9.32. An LR(2) Grammar

with a `q` but the parser cannot see if it should reduce by `C->q` (look-ahead `e`), reduce by `D->q` (look-ahead `e`), or shift over `e`. But each choice has a different two-token look-ahead set (`ec`, `ed` and {`ea`, `eb`}, respectively), so LR(2) should work.

The initial state, state 1, in the LR(2) parser for this grammar is

```
S->•Aa ##
S->•Bb ##
S->•Cec ##
S->•Ded ##
A->•qE a#
B->•qE b#
C->•q ec
D->•q ed
```

which calls for a shift over the `q`. After this shift the parser reaches a state

```
A→q•E a#
B→q•E b#
C→q• ec
D→q• ed
E→•e a#
E→•e b#
```

where we still have the same shift/reduce conflict: there are two reduce items, **C→q•** and **D→q•** with look-aheads **ec** and **ed**, and one shift item, **E→•e**, which shifts on an **e**.

The conflict goes away when we realize that for each item I two kinds of look-aheads are involved: the *item look-ahead*, the set of strings that can follow the end of I; and the *dot look-ahead*, the set of strings that can follow the dot in I. For parsing decisions it is the dot look-ahead that counts, since the dot position corresponds with the gap in an LR parser, so the dot look-ahead corresponds to the first k tokens of the rest of the input. Note that for reductions the item look-ahead seems to be the deciding factor, but since the dot is at the end in reduce items, the item look-ahead coincides with the dot look-ahead. In an LR(1) parser the dot look-ahead of a shift item I coincides with the set of tokens on which there is a shift from the state I resides in, so there is no need to compute it separately, but as we have seen above, this is not true for an LR(2) parser.

So we compute the full two-token dot look-aheads for the shift items to obtain state 2:

item with item look-ahead	dot look-ahead
A→q•E a#	**ea**
B→q•E b#	**eb**
C→q• ec	**ec**
D→q• ed	**ed**
E→•e a#	**ea**
E→•e b#	**eb**

Now the conflict is resolved since the two reduce actions and the shift action all have different dot look-aheads: shift on **ea** and **eb**, reduce to **C** on **ec**, and reduce to **D** on **ed**.

More in general, the dot look-ahead of an item $A \rightarrow \alpha\bullet\beta\ \gamma$, where γ is the item look-ahead, can be computed as $\text{FIRST}_k(\beta\gamma)$.

Parts of the ACTION and GOTO tables for the LR(2) parser for the grammar in Figure 9.32 are given in Figure 9.33. The ACTION table is now indexed by look-ahead strings of length 2 rather than by single tokens, but the GOTO table is still indexed by single symbols, since each entry in a GOTO table represents a transition in the handle-finding automaton, and transitions consume just one symbol. As a result, superimposing the two tables into one ACTION/GOTO table is no longer possible; combined ACTION/GOTO tables are a feature of LR(1) parsing only (and, with some handwaving, of LR(0)). Again all the error detection is done in the ACTION table.

ACTION

	qe	ea	eb	ec	ed	⋯
1	s	e	e	e	e	⋯
2	e	s	s	r4	r5	⋯
3	⋯					
4	⋯					
⋮	⋮					

GOTO

	q	a	b	c	d	e	E	⋯
1	2							⋯
2						3	4	⋯
3	⋯							
4	⋯							
⋮	⋮							

Fig. 9.33. Partial LR(2) ACTION and GOTO tables for the grammar of Figure 9.32

It is interesting to compare this to LR(0), where there is no look-ahead at all. There the ACTION table offers no protection against impossible shifts, and the GOTO table has to contain error entries. So we see that the LR(0), LR(1), and LR($k > 1$) table construction algorithms differ in more than just the value of k: LR(0) needs a check upon shift; LR($k > 1$) needs the computation of dot look-ahead; and LR(1) needs either but not both. It is of course possible to design a combined algorithm, but for all values of k part of it would not be activated.

However interesting LR($k > 1$) parsing may be, its practical value is quite limited: the required tables can assume gargantuan size (see, e.g., Ukkonen [66]), and it does not really help much. Although an LR(2) parser is more powerful than an LR(1) parser, in that it can handle some grammars that the other cannot, the emphasis is on "some". If a common-or-garden variety grammar is not LR(1), chances are minimal that it is LR(2) or higher.

9.6.3 Some Properties of LR(k) Parsing

Some theoretically interesting properties of varying practical significance are briefly mentioned here. It can be proved that any LR(k) grammar with $k > 1$ can be transformed into an LR($k-1$) grammar (and so to LR(1), but not always to LR(0)), often at the expense of an enormous increase in size; see for example Mickunas, et al. [407]. It can be proved that if a language allows parsing with a pushdown automaton as described in Section 3.3, it has an LR(1) grammar; such languages are called *deterministic languages*. It can be proved that if a grammar can be handled by any of the deterministic methods of Chapters 8 and 9, it can be handled by an LR(k) parser (that is, all deterministic methods are weaker than or equally strong as LR(k)). It can be proved that any LR(k) language can be obtained as a regular expression, the elements of which are LR(0) languages; see Bertsch and Nederhof [96].

LR($k \geq 1$) parsers have the immediate error detection property: they will stop at the first incorrect token in the input and not even perform another shift or reduce. This is important because this early error detection property allows a maximum amount of context to be preserved for error recovery; see Section 16.2.6. We have seen that LR(0) parsers do not have this property.

In summary, LR(k) parsers are the strongest deterministic parsers possible and they are the strongest linear-time parsers known, with the exception of some non-

canonical parsers; see Section 10. They react to errors immediately, are paragons of virtue and beyond compare, but even after 40 years they are not widely used.

9.7 LALR(1)

The reader will have sensed that our journey has not yet come to an end; the goal of a practical, powerful, linear-time parser has still not been attained completely. At their inception by Knuth in 1965 [52], it was realized that LR(1) parsers would be impractical in that the space required for their deterministic automata would be prohibitive. A modest grammar might already require hundreds of thousands or even millions of states, numbers that were totally incompatible with the computer memories of those days.

In the face of this difficulty, development of this line of parsers came to a standstill, partially interrupted by Korenjak's invention of a method to partition the grammar, build LR(1) parsers for each of the parts and combine these into a single over-all parser (Korenjak [53]). This helped, but not much, in view of the added complexity.

The problem was finally solved by using an unlikely and discouraging-looking method. Consider the LR(1) automaton in Figure 9.27 and imagine boldly discarding all look-ahead information from it. Then we see that each state in the LR(1) automaton reverts to a specific state in the LR(0) automaton; for example, LR(1) states 6 and 13 collapse into LR(0) state 6 and LR(1) states 2 and 9 collapse into LR(0) state 2. We say that LR(1) states 6 and 13 have the same *core*, the items in the LR(0) state 6, and similarly for LR(1) states 2 and 9.

There is not a single state in the LR(1) automaton that was not already present in a rudimentary form in the LR(0) automaton. Also, the transitions remain intact during the collapse: both LR(1) states 6 and 13 have a transition to state 9 on **T**, but so has LR(0) state 6 to 2. By striking out the look-ahead information from an LR(1) automaton, it collapses into an LR(0) automaton for the same grammar, with a great gain as to memory requirements but also at the expense of the look-ahead power. This will probably not surprise the reader too much, although a formal proof of this phenomenon is not trivial.

The idea is now to collapse the automaton but to keep the look-ahead information, as follows. The LR(1) state 2 (Figure 9.27) contains the items

```
E→T• -
E→T• #
```

and LR(1) state 9 contains

```
E→T• -
E→T• )
```

where the LR(0) core is

```
E→T•
E→T•
```

They collapse into an LALR(1) state which corresponds to the LR(0) state 2 in Figure 9.25, but now with look-ahead:

```
E→T• #
E→T• -
E→T• )
```

The surprising thing is that this procedure preserves almost all the original look-ahead power and still saves an enormous amount of memory. The resulting automaton is called an *LALR(1) automaton*, for "Look Ahead LR(0) with a look-ahead of 1 token."

The LALR(1) automaton for our grammar of Figure 9.23 is given in Figure 9.34. The look-aheads are sets now and are shown between `[` and `]`, so state 2 is repre-

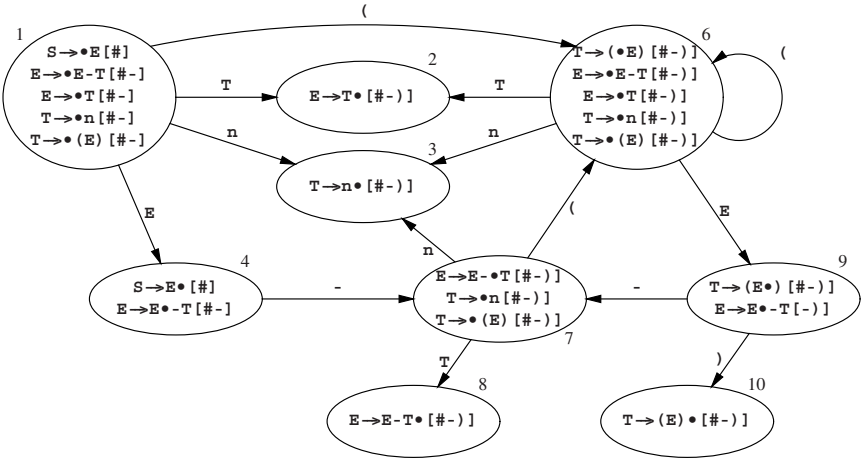


Fig. 9.34. The LALR(1) automaton for the grammar of Figure 9.23

sented as `E→T• [#-)]`. We see that the original conflict in state 4 is indeed still resolved, as it was in the LR(1) automaton, but that its size is equal to that of the LR(0) automaton. Now that is a very fortunate state of affairs!

We have finally reached our goal. LALR(1) parsers are powerful, almost as powerful as LR(1) parsers, they have fairly modest memory requirements, only slightly inferior to (= larger than) those of LR(0) parsers,[2] and they are time-efficient. LALR(1) parsing may very well be the most-used parsing method in the world today. Probably the most famous LALR(1) parser generators are *yacc* and its GNU version *bison*.

LALR(k) also exists and is LR(0) with an add-on look-ahead of k tokens. LALR(k) combines LR(0) information about the left context (in the LR(0) automa-

[2] Since the LALR(1) tables contain more information than the LR(0) tables (although they have the same size), they lend themselves slightly less well to data compression. So practical LALR(1) parsers will be bigger than LR(0) parsers.

ton) with LR(k) information about the right context (in the k look-aheads). Actually there is a complete family of *LA(k)LR(j)* parsers out there, which combines LR(j) information about the left context with LR(k) information about the right context. Like LALR(1), they can be derived from LR($j+k$) parsers in which all states with identical cores and identical first k tokens of the $j+k$-token look-ahead have coincided. So LALR(1) is actually LA(1)LR(0), Look-ahead Augmented (1) LR (0). See Anderson [55].

9.7.1 Constructing the LALR(1) Parsing Tables

When we have sufficiently drunk in the beauty of the vista that spreads before us on these heights, and start thinking about returning home and actually building such a parser, it will come to us that there is a small but annoying problem left. We have understood how the desired parser should look and also seen how to construct it, but during that construction we used the unacceptably large LR(1) parser as an intermediate step.

So the problem is to find a shortcut by which we can produce the LALR(1) parse table without having to construct the one for LR(1). This particular problem has fascinated scores of computer scientists for many years (see the references in (Web)Section 18.1.4), and several good (and some very clever) algorithms are known. On the other hand, several deficient algorithms have appeared in publications, as DeRemer and Pennello [63] and Kannapinn [99] have pointed out. (These algorithms are deficient in the sense that they do not work for some grammars for which the straightforward LR(1) collapsing algorithm does work, rather than in the sense that they would lead to incorrect parsers.)

Since LALR(1) is clearly a difficult concept; since we hope that each new LALR algorithm contributes to its understandability; and since we think some algorithms are just too interesting to skip, we have allowed ourselves to discuss four LALR(1) parsing table construction algorithms, in addition to the one above. We present 1. a very simple algorithm, which shows that constructing an LALR(1) parsing table is not so difficult after all; 2. the algorithm used in the well-known parser generator *yacc*; 3. an algorithm which creates LALR(1) by upgrading LR(0); and 4. one that does it by converting the grammar to SLR(1). This is also the order in which the algorithms were discovered.

9.7.1.1 A Simple LALR(1) Algorithm

The easiest way to keep the LALR(1) parse table small is to never let it get big. We achieve this by collapsing the states the moment they are created, rather than first creating all states and then collapsing them. We start as if we are making a full LR(1) parser, propagating look-aheads as described in Section 9.6, and we use the table building technique of Section 9.5.5. In this technique we create new states by performing transitions from existing unprocessed states obtained from a list U, and if the created state v is not already present in the list of processed states S, the algorithm adds it to U so it can be the source of new transitions.

For our LALR(1) algorithm we refine this step as follows. We check *v* to see if there is already a state *w* in *S* with the same core. If so, we merge *v* into *w*; if this modifies *w*, we put *w* back in *U* as an unprocessed state: since it has changed, it may lead to new and different states. If *w* was not modified, no new information has come to light and we can just extract the next unprocessed state from *U*; *v* itself is discarded in both cases. The state *w* keeps its number and its transitions; it is important to note that when *w* is processed again, its transitions are guaranteed to lead to states whose cores are already present in *S* and *T*.

The merging makes sure that the cores of all states in *S* are always different, as they should be in an LALR(1) parser; so never during the process will the table be larger than the final LALR(1) table. And by putting all modified states back into the list to be processed we have ensured that all states with their proper LALR(1) look-aheads will be found eventually. This surprisingly simple algorithm was first described by Anderson et al. [56] in 1973.

The algorithm is not ideal. Although it solves the main problem of LALR(1) parse table generation, excessive memory use, it still generates almost all LR(1) states, of which there are many more than LALR(1) states. The only situation in which we gain time over LR(1) parse table generation is when merging the created state *v* into an existing state *w* does not modify *w*. But usually *v* will bring new look-aheads, so usually *w* will change and will then be reprocessed. Computer scientists, especially compiler writers, felt the need for a faster LALR(1) algorithm, which led to the techniques described in the following three sections.

9.7.1.2 The Channel Algorithm

The well-known parser generator *yacc* uses an algorithm that is both intuitively relatively clear and reasonably efficient (Johnson [361]); it is described in more detail by Aho, Sethi and Ullman in [340]. The algorithm does not seem to have a name; we shall call it the *channel algorithm*.

We again use the grammar of Figure 9.23, which we now know is LALR(1) (but not LR(0)). Since we want to do look-ahead but do not yet know what to look for, we use LR(0) items extended with a yet unknown look-ahead field, indicated by an empty square; an example of an item would be **A→bC•De** □. Using such items, we construct the non-deterministic LR(0) automaton in the usual fashion; see Figure 9.35. Now suppose that we were told by some oracle what the look-ahead set of the item **S→•E** □ is (first column, second row in Figure 9.35); call this look-ahead set *L*. Then we could draw a number of conclusions. The first is that the item **S→E•** □ also has *L*. The next is that the look-ahead set of the station **•E**□ is also *L*, and from there *L* spreads to **E→•E-T**, **E→E•-T**, **E→E-•T**, **E→E-T•**, **E→•T** and **E→T•**. From **E→E-•T** and **E→•T** it flows to the station **•T** and from there it again spreads on.

The flow possibilities of look-ahead information from item to item once it is known constitute "channels" which connect items. Each channel connects two items and is one-directional. There are two kinds of channels. From each station channels run down to each item that derives from it; these channels propagate input from

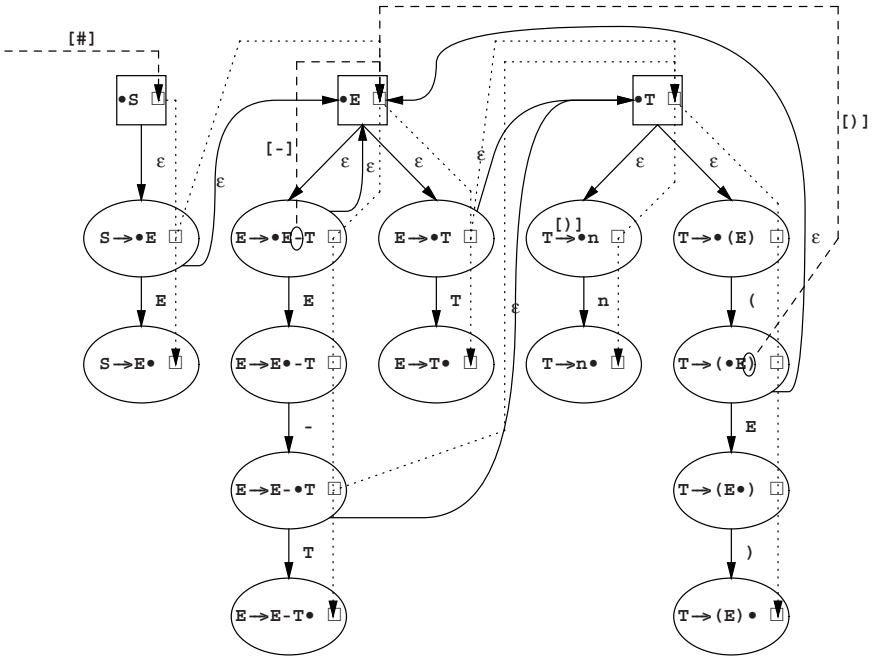


Fig. 9.35. Non-deterministic automaton with channels

elsewhere. From each item that has the dot in front of a non-terminal *A*, a channel runs parallel to the ε-arrow to the station •*A*□. If *A* is the last symbol in the right-hand side, the channel propagates the look-ahead of the item it starts from. If *A* is not the last symbol, but is followed by, for example, *CDe* (so the entire item would be something like $P \rightarrow B \bullet ACDe$ □), the input to the channel is FIRST(*CDe*); such input is said to be "generated spontaneously", as opposed to "propagated" input.

Figure 9.35 shows the full set of channels: those carrying propagated input as dotted lines, and those carrying spontaneous input as dashed lines, with their spontaneous input sets. A channel from outside introduces the spontaneous look-ahead **#**, the end-of-input marker, to the station(s) of the start symbol. The channel set can be represented in a computer as a list of input and output ends of channels:

Input end	leads to	output end	Remarks
[#]	==>	•S□	spontaneous
•S□	==>	S→•E □	propagated
S→•E □	==>	S→E• □	propagated
S→•E □	==>	•E □	propagated
	...		
[-]	==>	•E □	spontaneous
	...		

Next we run the subset algorithm on this (channeled) NFA in slow motion and watch carefully where the channels go. This procedure severely taxes the human

brain; a more practical way is to just construct the deterministic automaton without concern for channels and then use the above list (in its complete form) to re-establish the channels. This is easily done by finding the input and output end items and stations in the states of the deterministic automaton and constructing the corresponding channels. Note that a single channel in the NFA can occur many times in the deterministic automaton, since items can (and will) be duplicated by the subset construction. The result can best be likened to a bowl of mixed spaghetti and tagliatelli (the channels and the transitions) with occasional chunks of ham (the item sets) and will not be printed in this book.

Now we are close to home. For each channel we pump its input to the channel's end. First this will only have effect for channels that have spontaneous input: a **#** will flow in state 1 from item **S→•E[□]** to station **•E[□]**, which will then read **•E[#]**; a **-** from **E→•E-T[□]** flows to the **•E[□]**, which changes to **•E[-]**; etc. We go on pumping until all look-ahead sets are stable and nothing changes any more. We have now obtained the LALR(1) automaton and can discard the channels; of course we keep the transitions. This is an example of a transitive closure algorithm.

It is interesting to look more closely at state 4 (see Figure 9.34) and to see how **S→E•[#]** gets its look-ahead which excludes the **-**, although the **-** is present in the look-ahead set of **E→E•-T[#-]** in state 4. To this end, a magnified view of the top left corner of the full channeled LALR(1) automaton is presented in Figure 9.36; it comprises the states 1 to 4. Again channels with propagated input are dotted, those with spontaneous input are dashed and transitions are drawn. We can now see more clearly that **S→E•[#]** derives its look-ahead from **S→•E[#]** in 1, while **E→E•-T[#-]** derives its look-ahead (indirectly) from **•E[-]** in state 1. This item has a look-ahead **-** generated spontaneously in **E→•E-T[□]** in state 1. The channel from **S→•E[#]** to **•E[#-]** only works "downstream", which prevents the **-** from flowing back. LALR(1) parsers often give one the feeling that they succeed by a narrow margin!

If the grammar contains ε-rules, the same complications arise as in Section 9.6.1 in the determination of the FIRST set of the rest of the right-hand side: when a non-terminal is nullable we have to also include the FIRST set of what comes after it, and so on. We meet a special complication if the entire rest of the right-hand side can be empty: then we may see the look-ahead □, which we do not know yet. In fact this creates a third kind of channel that has to be watched in the subset algorithm. We shall not be so hypocritical as to suggest the construction of the LALR(1) automaton for the grammar of Figure 9.30 as an exercise to the reader, but we hope the general principles are clear. Let a parser generator do the rest.

9.7.1.3 LALR(1) by Upgrading LR(0)

The above techniques basically start from an LR(1) parse table, explicit or implicit, and then shrink it until the items are LR(0): they downgrade the LR(1) automaton to LALR(1). It is also possible to start from the LR(0) automaton, find the conflicts in it, and upgrade from there. This leads to a complicated but very efficient algorithm,

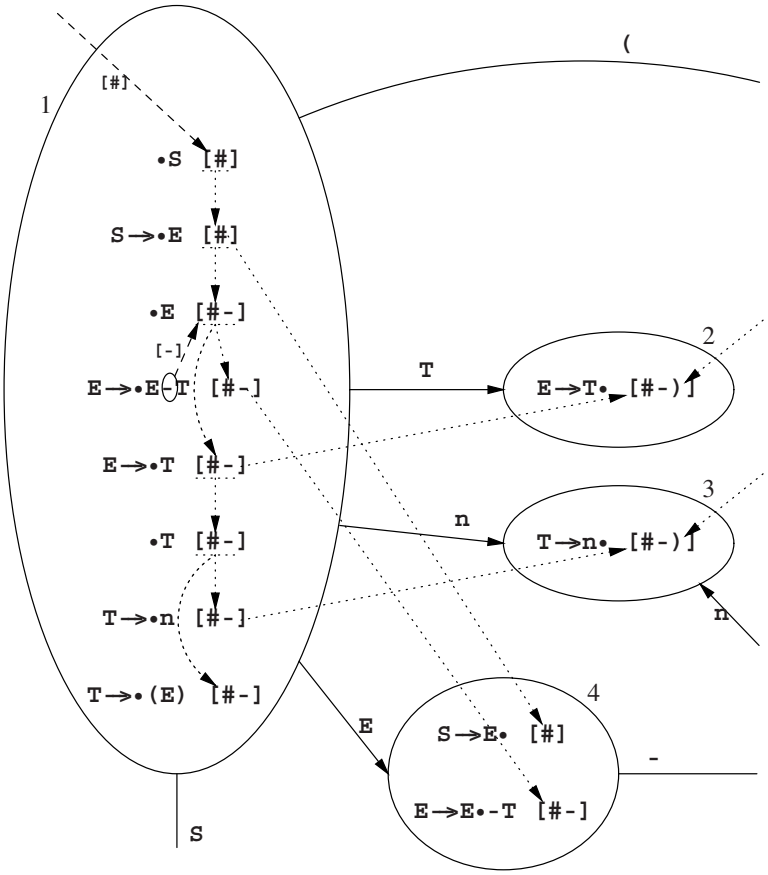


Fig. 9.36. Part of the deterministic automaton with channels (magnified cut)

designed by DeRemer and Pennello [63]. Again it has no name; we shall call it the *relations algorithm*, for reasons that will become clear.

Upgrading the inadequate LR(0) automaton in Figure 9.25 is not too difficult. We need to find the look-ahead(s) we are looking at in the input when we are in state 4 and reducing by **S→E** is the correct action. That means that the stack must look like

··· **E** ④

Looking back through the automaton, we can see that we can have come from one state only: state 1:

① **E** ④

Now we do the reduction because we want to see what happens when that is the correct action:

① **S**

and we see that we have reduced to `S`, which has only one look-ahead, `#`, the end-of-input token. So the reduce look-ahead of the item `S→E•` in state 4 is `#`, which differs from the shift look-ahead `-` for `E→E•-T`, so the conflict is resolved.

This is an example of a more general technique: to find the look-ahead(s) of an inadequate reduce item in an LR(0) automaton, we take the following steps:

- we assume that the implied reduction R is the proper action and simulate its effects on an imaginary stack;
- we simulate all possible further movements of the LR(0) automaton until the automaton is about to shift in the next token, t;
- we add t to the look-ahead set, since it has the property that it will be shifted and accepted if we do the reduction R when we see it as the next token in the input.

It will be clear that this is a very reasonable method of collecting good look-ahead sets. It is much less clear that it produces the same LALR look-ahead sets as the LALR algorithms above, and for a proof of that fact we refer the reader to DeRemer and Pennello's paper.

Turning the above ideas into an algorithm requires some serious effort. We will follow DeRemer and Pennello's explanation closely, using the same formalism and terminology as much as is convenient. The explanation uses an unspecified grammar of which only two rules are important: $A \rightarrow \omega$ and $B \rightarrow \beta A \gamma$, for some, possibly empty sequences of non-terminals and terminals ω, β, and γ. Refer to Figure 9.37.

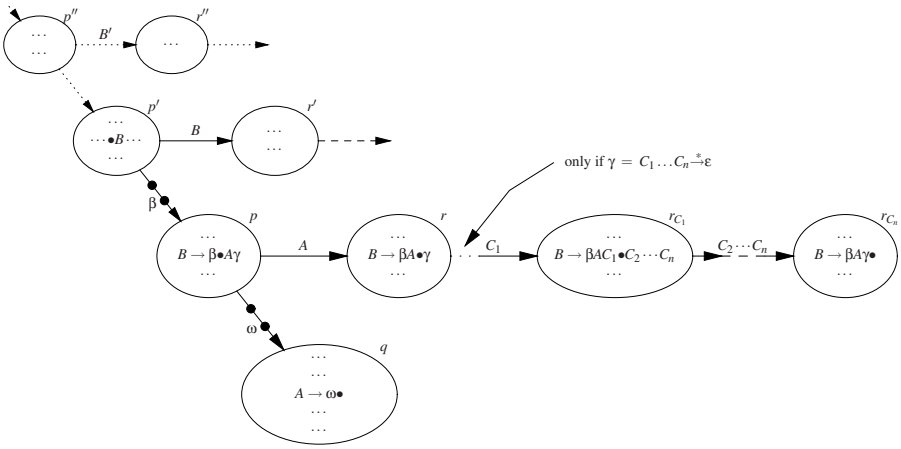


Fig. 9.37. Hunting for LALR(1) look-aheads in an LR(0) automaton — the **lookback** and **includes** relations

Suppose the LR(0) automaton has an inadequate state q with a reduce item $A \rightarrow \omega\bullet$, and we want to know the LALR look-ahead of this item. If state q is on the top of the stack, there must be a path through the LR(0) automaton from the start state 1 to q (or we would not have ended up in q), and the last part of this path spells ω (or we would not be able to reduce by $A \rightarrow \omega$). We can follow this path back to the

beginning of ω; this leads us to the state p, where the present item $A \rightarrow \omega\bullet$ originated. There are two things to note here: there may be several different paths back that spell ω, leading to several different ps; and ω may be ε, in which case p is equal to q. For simplicity Figure 9.37 shows one p only.

We have now established that the top segment of the stack is $p\ \omega_1\ \cdots\ \omega_n\ q$, where p is one of the ps identified above and $\omega_1 \cdots \omega_n$ are the components of ω. We can now do the simulated reduction, as we did above. This shortens the stack to $p\ A$, and we have to shift over the A, arriving at a state r.

More formally, a reduce item $A \rightarrow \omega\bullet$ in an LR(0) state q identifies a set of transitions $\{p_1 \xrightarrow{A} r_1, \ldots, p_n \xrightarrow{A} r_n\}$, where for all p_i we have $p_i \xrightarrow{\omega} q$. This defines the so-called **lookback** relation between a pair (state, reduce item) and a transition. One writes $(q,\ A \rightarrow \omega\bullet)$ **lookback** $(p_i \xrightarrow{A} r_i)$ for $1 \leq i \leq n$. This is step 1 of the simulation. Note that this is a relation, not an algorithm to compute the transition(s); it just says that given a pair (state, reduce item) and a transition, we can check if the "lookback" relation holds between them. (DeRemer and Pennello write a transition $(p_i \xrightarrow{A} r_i)$ as (p_i, A), since the r follows directly from the LR(0) automaton, which is deterministic.)

The shift from p over A is guaranteed to succeed, basically because the presence of an item $A \rightarrow \omega\bullet$ in q combined with the existence of a path ω from q leading back to p proves that p contains an item that has a dot in front of an A. That $\bullet A$ causes both the ω path and the transition $p \xrightarrow{A} r$ (except when A is the start symbol, in which case we are done and the look-ahead is **#**). The general form of such an item is $B \rightarrow \beta\bullet A\gamma$, as shown in Figure 9.37. Here we have the first opportunity to see some look-ahead tokens: any terminal in FIRST(γ) will be an LALR look-ahead token for the reduce item $A \rightarrow \omega\bullet$ in state q. But the simulation is not finished yet, since γ may be or produce ε, in which case we will also have to look past the item $B \rightarrow \beta A\bullet\gamma$.

If γ produces ε, it has to consist of a sequence of non-terminals $C_1 \cdots C_n$, each capable of producing ε. This means that state r contains an item $C_1 \rightarrow \bullet$, which is immediately a reduce item; see a similar phenomenon in state 3 in Figure 9.31. Its presence will certainly make r an inadequate state, but, if the grammar is LALR(1), that problem will be solved when the algorithm treats the item $C_1 \rightarrow \bullet$ in r. For the moment we assume the problem is solved; we do the reduction, push C_1 on the simulated stack, and shift over it to state r_{C_1}. We repeat this process until we have processed all $C_1 \cdots C_n$, and by doing so reach a state r_{C_n} which contains a reduce item $B \rightarrow \beta A\gamma\bullet$.

Now it is tempting to say that any look-ahead of this item will also figure in the look-ahead that we are looking for, but that is not true. At this point in our simulation the stack contains $p\ A\ r\ C_1\ r_{C_1}\ \cdots\ C_n\ r_{C_n}$, so we see only the look-aheads of those items $B \rightarrow \beta A\gamma\bullet$ in state r_{C_n} that have reached that state through p! State r_{C_n} may be reachable through other paths, which may quite well bring in other look-aheads for the reduce item $B \rightarrow \beta A\gamma\bullet$ which do not belong in the look-ahead set of $A \rightarrow \omega$. So to simulate the reduction $B \rightarrow \beta A\gamma$ we walk the path γ back through the LR(0) automaton to state p, all the while removing C_is (components of γ) from the stack. Then from state p backwards we can freely find all paths that spell β, to reach all

states p'_i that contain the item $B \to \bullet\beta A\gamma$. Each of these states p' has a transition on B, for the same reasons p had a transition on A (again except when B is the start symbol). The transition over B leads to a state r', which brings us back to a situation similar to the one at p.

This process defines the so-called **includes** relation: $(p \xrightarrow{A} r)$ **includes** $(p' \xrightarrow{B} r')$ if and only if the grammar contains a rule $B \to \beta A\gamma$, and $\gamma \xrightarrow{*} \varepsilon$, and $p' \xrightarrow{\beta} p$. Note that one $(p \xrightarrow{A} r)$ can include several $(p' \xrightarrow{B} r')$s, when several paths β are possible.

To simulate all possible movements of the LR(0) automaton and find all the transitions that lead to states that contribute to the look-ahead of $A \to \omega\bullet$ in state q, we have to repeat the step from p to p' for successive p'', p''', ..., until we find no new ones any more or until we are stopped by reaching a reduction of the start symbol. This is step 2 of the simulation.

Any token t that can be shifted over in any of the states r, r', ... thus reached, belongs in the look-ahead of $A \to \omega\bullet$ in state q, since we have just shown that after the reduction $A \to \omega$ and possibly several other reductions, we arrive at a state in which a shift over t is possible. And no other tokens belong in the look-ahead set, since they will not allow a subsequent shift, and would get the parser stuck.

So we are interested in the terminal transitions of the states r, r', ... To describe them in a framework similar to the one used so far, we define a relation **directly-reads** as follows; refer to Figure 9.38. A transition $(p \xrightarrow{A} r)$ **directly-reads** t if r has

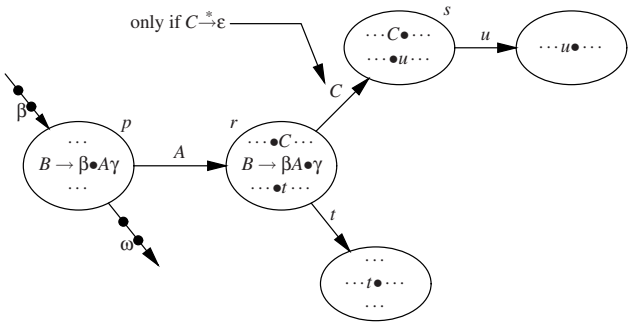


Fig. 9.38. Hunting for LALR(1) look-aheads in an LR(0) automaton — the **directly-reads** and **reads** relations

an outgoing arrow on the terminal symbol t. Actually, neither p nor A is used in this definition, but we start from the transition $p \xrightarrow{A} r$ rather than from the state r because the **lookback** and **includes** relations use transitions rather than states.

Again nullable non-terminals complicate the situation. If r happens to have an outgoing arrow marked with a non-terminal C that produces ε, we can reduce ε to C in our simulation, stack it, shift over it and reach another state, say s. Then anything we are looking at after the transition $r \xrightarrow{C} s$ must also be added to the look-ahead set of $A \to \omega\bullet$. Note that this C need not be the C_1 in Figure 9.38; it can be any

nullable non-terminal marking an outgoing arrow from state r. This defines the **reads** relation: $(p \xrightarrow{A} r)$ **reads** $(r \xrightarrow{C} s)$ if and only if both transitions exist and $C \xrightarrow{*} \varepsilon$. And then all tokens u that fulfill $(r \xrightarrow{C} s)$ **directly-reads** u belong in the look-ahead set of $A \rightarrow \omega\bullet$ in state q. Of course state s can again have transitions on nullable non-terminals, which necessitate repeated application of the "**reads** and **directly-reads**" operation. This is step 3 of the simulation.

We are now in a position to formulate the LALR look-ahead construction algorithm in one single formula. It uses the final relation in our explanation, **in-LALR-lookahead**, which ties together a reduce item in a state and a token: t **in-LALR-lookahead** $(q, A \rightarrow \omega\bullet)$, with the obvious meaning. The relations algorithm can now be written as:

$$
\begin{aligned}
t \textbf{ in-LALR-lookahead } (q, A \rightarrow \omega\bullet) \quad = \\
\quad (q, A \rightarrow \omega\bullet) \textbf{ lookback } (p \xrightarrow{A} r) \textbf{ includes } (p' \xrightarrow{B} r') \cdots \\
\quad \cdots \textbf{ includes } (p'' \xrightarrow{B'} r'') \textbf{ reads } (r'' \xrightarrow{C} s) \cdots \\
\quad \cdots \textbf{ reads } (r''' \xrightarrow{C'} s') \textbf{ directly-reads } t
\end{aligned}
$$

This is not a formula in the arithmetic sense of the word: one cannot put in parentheses to show the precedences, as one can in $a + b \times c$; it is rather a linked sequence of relations, comparable to $a < b \leq c < d$, in which each pair of values must obey the relational operator between them. It means that a token t is in the LALR lookahead set of reduce item $A \rightarrow \omega\bullet$ in state q if and only if we can find values for p, p', ..., B, B', ..., r, r', ..., C, C', ..., and s, s', ..., so that all the relations are obeyed.

In summary, when you do a reduction using a reduce item, the resulting non-terminal either is at the end of another item, in which case you have to include that item in your computations, or it has something in front of it, in which case your look-ahead set contains everything you can read from there, directly or through nullable non-terminals.

The question remains how to utilize the sequence of relations to actually compute the LALR look-ahead sets. Two techniques suggest themselves. We can start from the pair $(q, A \rightarrow \omega\bullet)$, follow the definitions of the relations until we reach a token t, record it, backtrack and exhaustively search all possibilities: the top-down approach. We can also make a database of relation triples, insert the initially known triples and apply the relation definitions until nothing changes any more: the transitive closure approach. Both have their problems. The top-down method has to be careful to prevent being caught in loops, and will often recompute relations. The transitive closure sweep will have to be performed an indefinite number of times, and will compute triples that do not contribute to the solution.

Fortunately there is a better way. It is not immediately evident, but the above algorithm has a remarkable property: it only uses the grammar and the LR(0) transitions over non-terminals (except for both ends of the relation sequence); it never looks inside the LR(0) states. The reasonings that show the validity of the various definitions use the presence of certain items, but the final definitions do not. This makes it particularly easy to express the relations as arcs in a directed graph in which the non-terminal transitions are the nodes.

The relations graph corresponding to Figures 9.37 and 9.38 is shown in Figure 9.39. We see that it is quite different from the transition graphs in Figures 9.37 and

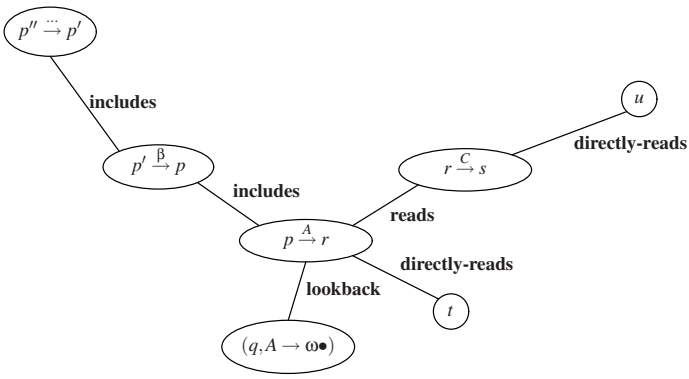


Fig. 9.39. Hunting for LALR(1) look-aheads in an LR(0) automaton — the relations graph

9.38: the transition arcs in those graphs have become nodes in the new graph, and the relations, not present in the old graphs, are the arcs in the new one. To emphasize this fact, the transition nodes in Figure 9.39 have been drawn in the same relative positions as the corresponding arcs in Figures 9.37 and 9.38; this is the cause of the strange proportions of Figure 9.39.

The LALR look-ahead sets can now be found by doing a transitive closure on this graph, to find all leaves connected to the $(q, A \rightarrow \omega\bullet)$ node. The point is that there exists a very efficient algorithm for doing transitive closure on a graph, the "SCCs algorithm". This algorithm successively isolates and condenses "strongly connected components" of the graph; hence its name. The algorithm was invented by Tarjan [334] in 1972, and is discussed extensively in books on algorithms and on the Internet.

DeRemer and Pennello describe the details required to cast the sequence of relations into a graph suitable for the SCCs algorithm. This leads to one of the most efficient LALR parse table construction algorithms known. It is linear in the number of relations involved in the computation, and in practice it is linear in the number of non-terminal transitions in the LR(0) automaton. It is several times faster than the channel algorithm used in *yacc*. Several optimizations can be found (Web)Section 18.1.4. Bermudez and Schimpf [76] extend the algorithm to LALR(k).

When reaching state r_{C_n} in Figure 9.37 we properly backtracked over all components of γ back to state p, to make sure that all look-aheads found could indeed be shifted when we perform the reduction $A \rightarrow \omega$. If we omit this step and just accept any look-ahead at r_{C_n} as look-ahead of $A \rightarrow \omega$, we obtain an *NQLALR(1)* parser, for "Not Quite LALR(1)". NQLALR(1) grammars are strange in that they do not fit in the usual hierarchy ((Bermudez and Schimpf [75]); but then, that can be expected from an incorrect algorithm.

9.7.1.4 LALR(1) by Converting to SLR(1)

When we look at the non-LR(0) automaton in Figure 9.25 with an eye to upgrading it to LALR(1), we realize that, for example, the **E** along the arrow from state 1 to state 4 is in fact a different **E** from that along the arrow from state 6 to state 9, in that it arises from a different station •**E**, the one in state 1, and it is the station that gets the look-ahead. So to distinguish it we can call it ①**E**④, so now the item **S**→•**E** reads **S**→•①**E**④, where ①**E**④ is just a non-terminal name, in spite of its appearance. This leads to the creation of a station •①**E**④ (not shown) which produces two items based on the two rules **E**→**T** and **E**→**E-T**. We can even give the non-terminals in these rules more specific names:

①**E**④ → ①**T**②
①**E**④ → ①**E**④ ④-⑦ ⑦**T**⑧

where we obtained the other state numbers by following the rules through the LR(0) automaton.

Continuing this way we can construct an "LR(0)-enhanced" version of the grammar of Figure 9.23; it is shown in Figure 9.40. A grammar rule $A \rightarrow BcD$ is trans-

①**S**◇ → ①**E**④
①**E**④ → ①**E**④ ④-⑦ ⑦**T**⑧ | ①**T**②
⑥**E**⑨ → ⑥**E**⑨ ⑨-⑦ ⑦**T**⑧ | ⑥**T**②
①**T**② → ①**n**③
⑥**T**② → ⑥**(**⑥ ⑥**E**⑨ ⑨**)**⑩ | ⑥**n**③
⑦**T**⑧ → ⑦**(**⑥ ⑥**E**⑨ ⑨**)**⑩ | ⑦**n**③

Fig. 9.40. An LR(0)-enhanced version of the grammar of Figure 9.23

formed into a new grammar rule $(s_1)A(s_x) \rightarrow (s_1)B(s_2)\ (s_2)c(s_3)\ (s_3)D(s_4)$, where (s_x) is the state shifted to by the non-terminal, and $(s_1)\cdots(s_4)$ is the sequence of states met when traveling down the right-hand side of the rule in the LR(0) automaton.

We see that the rules for **E** have been split into two versions, one starting at ① and the other at ⑥, and likewise the rules for **T**. It is clear that the look-aheads of the station •①**E**④ all end up in the look-ahead set of the item **E**→**E-T**• reached at the end of the sequence ①**E**④ ④-⑦ ⑦**T**⑧, so it is interesting to find out what the look-ahead set of the •①**E**④ in state 1 is, or rather just what the look-ahead set of •①**E**④ is, since there is only one •①**E**④ and it is in state 1.

Bermudez and Logothetis [79] have given a surprisingly simple answer to that question: the look-ahead set of •①**E**④ is the FOLLOW set of ①**E**④ in the LR(0)-enhanced grammar, and likewise for all the other LR(0)-enhanced non-terminals. Normally FOLLOW sets are not very fine tools, since they combine the tokens that can follow a non-terminal N from all over the grammar, regardless of the context in which the production N occurs. But here the LR(0) enhancement takes care of the context, and makes sure that terminal productions of •**E** in state 1 are recognized

only if they really derive from ①**E**④. That all this leads precisely to an LALR(1) parser is less clear; for a proof see the above paper.

To resolve the inadequacy of the automaton in Figure 9.25 we want to know the look-ahead set of the item **S→E•** in state 4, which is the FOLLOW set of ①**S**◊. The FOLLOW sets of the non-terminals in the LR(0)-enhanced grammar are as follows:

FOLLOW(①**S**◊) = **[#]**
FOLLOW(①**E**④) = **[#-]**
FOLLOW(⑥**E**⑨) = **[-)]**
FOLLOW(①**T**②) = **[#-]**
FOLLOW(⑥**T**②) = **[-)]**
FOLLOW(⑦**T**⑧) = **[#-)]**

so the desired LALR look-ahead set is **#**, in conformance with the "real" LALR automaton in Figure 9.34. Since state 4 was the only inadequate state, no more look-aheads sets need to be computed.

Actually, the reasoning in the previous paragraph is an oversimplification: a reduce item in a state may derive from more than one station and import look-aheads from each of them. To demonstrate this we compute the look-aheads of **E→E-T•** in state 8. The sequence ends in state 8, so we select from the LR(0)-enhanced grammar those rules of the form **E→E-T** that end in state 8:

①**E**④ → ①**E**④ ④-⑦ ⑦**T**⑧
⑥**E**⑨ → ⑥**E**⑨ ⑨-⑦ ⑦**T**⑧

We see that the look-aheads of both stations •①**E**④ and •⑥**E**⑨ end up in state 8, and so the LALR look-ahead set of **E→E-T•** in that state is

FOLLOW(①**E**④) ∪ FOLLOW(⑥**E**⑨) = **[#-]** ∪ **[-)]** = **[#-)]**

Since this is the same way as look-aheads are computed in an SLR parser for a normal — not LR(0)-enhanced — grammar (Section 9.8), the technique is often referred to as "converting to SLR".

The *LALR-by-SLR* technique is algorithmically very simple:

- deriving the LR(0)-enhanced grammar from the original grammar and the LR(0) automaton is straightforward;
- computing the FOLLOW sets is done by a standard algorithm;
- selecting the appropriate rules from the LR(0)-enhanced grammar is simple;
- uniting the results is trivial.

And, as said before, only the look-ahead sets of reduce items in inadequate states need to be computed.

9.7.1.5 Discussion

LALR(1) tables can be computed by at least five techniques: collapsing and downgrading the LR(1) tables; Anderson's simple algorithm; the channel algorithm; by upgrading the LR(0) automaton; and by converting to SLR(1). Of these, Anderson's algorithm [56] (Section 9.7.1.1) is probably the easiest to program, and its

non-optimal efficiency should only seldom be a problem on present-day machines. DeRemer and Pennello [63]'s relations algorithm (Section 9.7.1.3) and its relatives discussed in (Web)Section 18.1.4 are among the fastest. Much technical and experimental data on several LALR algorithms is given by Charles [88].

Vilares Ferro and Alonso Pardo [372] describe a remarkable implementation of an LALR parser in Prolog.

9.7.2 Identifying LALR(1) Conflicts

When a grammar is not LR(1), the constructed LR(1) automaton will have conflicts, and the user of the parser generator will have to be notified. Such notification often takes such forms as:

> Reduce/reduce conflict
> in state 213 on look-ahead '`;`'
> `S→E` versus `A→T+E`

This may seem cryptic but the user soon learns to interpret such messages and to reach the conclusion that indeed "the computer can't see this". This is because LR(1) parsers can handle all deterministic grammars and our idea of "what a computer can see" coincides reasonably well with what is deterministic.

The situation is worse for those (relatively rare) grammars that are LR(1) but not LALR(1). The user never really understands what is wrong with the grammar: the computer should be able to make the right parsing decisions, but it complains that it cannot. Of course there is nothing wrong with the grammar; the LALR(1) method is just marginally too weak to handle it.

To alleviate the problem, some research has gone into methods to elicit from the faulty automaton a possible input string that would bring it into the conflict state. See DeRemer and Pennello [63, Sect. 7]. The parser generator can then display such input with its multiple partial parse trees.

9.8 SLR(1)

There is a simpler way to proceed with the NFA of Figure 9.35 than using the channel algorithm: first pump around the look-ahead sets until they are all known and then apply the subset algorithm, rather than vice versa. This gives us the so called *SLR(1)* automaton (for Simple LR(1)); see DeRemer [54]. The same automaton can be obtained without using channels at all: construct the LR(0) automaton and then add to each item $A \rightarrow \cdots$ a look-ahead set that is equal to FOLLOW(A). Pumping around the look-ahead sets in the NFA effectively computes the FOLLOW sets of each non-terminal and spreads these over each item derived from it.

The SLR(1) automaton is shown in Figure 9.41. Since FOLLOW(`S`)={`#`}, FOLLOW(`E`)={`#`,`-`,`)`} and FOLLOW(`T`)={`#`,`-`,`)`}, only states 1 and 4 differ from those in the LALR(1) automaton of Figure 9.34. The increased look-ahead sets do not spoil the adequateness of any states: the grammar is also SLR(1).

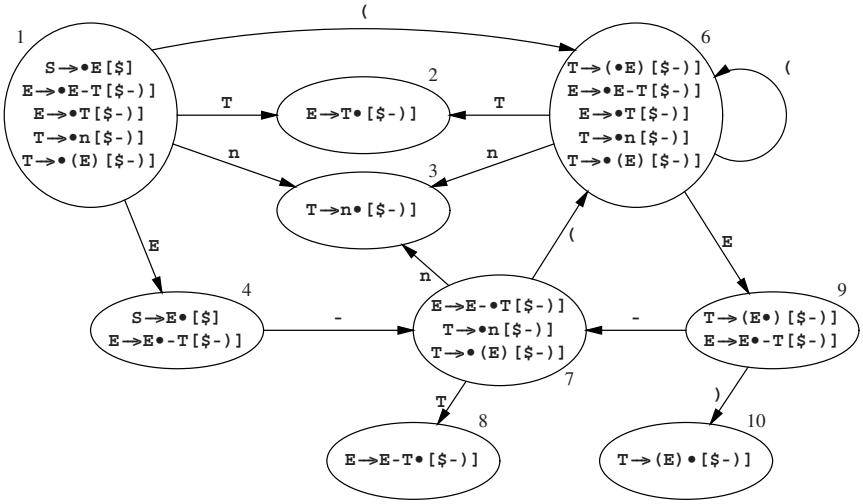


Fig. 9.41. SLR(1) automaton for the grammar of Figure 9.23

SLR(1) parsers are intermediate in power between LR(0) and LALR(1). Since SLR(1) parsers have the same size as LALR(1) parsers but are considerably less powerful, LALR(1) parsers are generally preferred.

FOLLOW$_k$ sets with $k > 1$ can also be used, leading to *SLR($k > 1$)* parsers. As with LA(k)LR(j), an LR(j) parser can be extended with additional FOLLOW$_k$ look-ahead, leading to *S(k)LR(j)* parsers. So SLR(1) is actually S(1)LR(0), and is just the most prominent member of the S(k)LR(j) parser family. To top things off, Bermudez and Schimpf [76] show that there exist NQSLR($k > 1$) parsers, thereby proving that "Simple LR" parsers are not really that simple for $k > 1$.

9.9 Conflict Resolvers

When states in an automaton have conflicts and no stronger method is available, the automaton can still be useful, provided we can find other ways to resolve the conflicts. Most LR parser generators have built-in conflict resolvers that will make sure that a deterministic automaton results, whatever properties the input grammar may have. Such a system will just enumerate the problems it has encountered and indicate how it has solved them.

Two useful and popular rules of thumb to solve LR conflicts are:

- on a shift/reduce conflict, shift (only on those look-aheads for which the conflict occurs);
- on a reduce/reduce conflict, reduce using the longest rule.

Both rules implement the same idea: take the largest bite possible. If you find that there is a production of A somewhere, make it as long as possible, including as much material on both sides as possible. This is very often what the grammar writer wants.

Systems with built-in conflict resolvers are a mixed blessing. On the one hand they allow very weak or even ambiguous grammars to be used (see for example, Aho, Johnson and Ullman [335]). This can be a great help in formulating grammars for difficult and complex analysis jobs; see, for example, Kernighan and Cherry [364], who make profitable use of automatic conflict resolution for the specification of typesetter input.

On the other hand a system with built-in conflict resolvers may impose a structure on the input where there is none. Such a system no longer corresponds to any grammar-like sentence-generating mechanism, and it may be very difficult to specify exactly what strings will be accepted and with what structure. How severe a drawback this is depends on the application and of course on the capabilities of the parser generator user.

It is to a limited extent possible to have dynamic (parse-time) conflict resolvers, as in the LL case (Section 8.2.5.3). Such a conflict resolver is called in a context that is still under construction, which complicates its use, but in simple cases its working can be understood and predicted. McKenzie [86] describes an extension of *yacc* that supports dynamic conflict resolvers, among other things.

Some experiments have been made with interactive conflict resolvers, which consult the user of the parser when a conflict actually arises: a large chunk of text around the conflict point is displayed and the user is asked to resolve the conflict. This is useful in, for example, document conversion; see Share [365].

9.10 Further Developments of LR Methods

Although the LALR(1) method as explained in Section 9.7 is quite satisfactory for most applications, a number of extensions to and improvements of the LR methods have been studied. The most important of these will be briefly explained in this section; for details see the literature, (Web)Section 18.1.4 and the original references.

For methods to speed up LR parsing by producing executable parser code see Section 17.2.2.

9.10.1 Elimination of Unit Rules

Many rules in practical grammars are of the form $A \rightarrow B$; examples can be found in Figures 2.10, 4.6, 5.3, 7.8, 8.7, 9.42 and many others. Such rules are called unit

```
Metre -> Iambic | Trochaic | Dactylic | Anapestic
```

Fig. 9.42. A (multiple) unit rule

rules, single rules, or chain rules. They generally serve naming purposes only and have no semantics attached to them. Consequently, their reduction is a matter of stack manipulation and state transition only, to no visible purpose for the user. Such

"administrative reductions" can take a considerable part of the parsing time (50% is not unusual). Simple methods to short-cut such reductions are easily found (for example, removal by systematic substitution) but may result in an exponential increase in table size. Better methods were found but turned out to be complicated and to impair the error detection properties of the parser. That problem can again be corrected, at the expense of more complication. See Heilbrunner [64] for a thorough treatment and Chapman [71] for much practical information.

Note that the term "elimination of unit rules" in this case is actually a misnomer: the unit rules themselves are not removed from the grammar, but rather their effect from the parser tables. Compare this to the actual elimination of unit rules in Section 4.2.3.2.

Actually unit rule elimination is a special case of stack activity reduction, which is discussed in the next section. But it was recognized earlier, and a separate body of literature exists for it.

9.10.2 Reducing the Stack Activity

Consider the stack of an LR parser, and call the state on top of the stack s_t. Now we continue the parser one step with proper input and we suppose this step stacks a token X and another state s_u, and we suppose that $s_u \neq s_t$, as will normally happen. Now, rather than being satisfied with the usual top stack segment $s_t\ X\ s_u$, we collapse this into one new state, $s_t + s_u$, which now replaces the original s_t. This means two things. First, we have lost the symbol X, and with it the possibility to construct a parse tree, so we are back to constructing a recognizer. But second, and more importantly, we have replaced an expensive stacking operation by a cheap state transition.

We can repeat this process of appending new states to the top state until one of two things happens: a state already in it is appended for the second time, or the original state s_t gets popped and we are left with an empty state. Only at that moment do we resume the normal stacking and unstacking operation of an LR parser.

When doing so for all acceptable inputs, we meet all kinds of compound states, all with s_t on the left, and many pairs are connected by transitions on symbols, terminal and non-terminal ones. Together they form a finite-state automaton. When we are forced to resume normal LR operation, it is very likely that we will find a state different from s_t on top, say s_x. We can then repeat the process for s_x and obtain another FSA.

Continuing this way we obtain a set of FSAs connected by stacking and unstacking LR operations. Using these FSAs instead of doing all the stack manipulation hidden in them greatly reduces the stack activity of the parser. Such a parser is called *reduction-incorporated* (*RI*).

In a traditional LR parser the gain in speed will almost certainly be outweighed by the disadvantage of not being able to construct a parse tree. Its great advantage lies in situations in which stack activity is expensive. Examples are the use of an LR parser as a subparser in a GLR parser (Chapter 11), where stack activity involves graph manipulation, and in parallel parsing (Chapter 14), where stack activity may require process communication.

The details of the algorithm are pretty complicated; descriptions are given by Aycock and Horspool [176] and Scott and Johnstone [100]. The resulting tables can be very large, even for every-day grammars.

9.10.3 Regular Right Part Grammars

As shown in Section 2.3.2.4, there are two interpretations of a regular right-hand side of a rule: the recursive and the iterative interpretation. The recursive interpretation is no problem: for a form like A^+ anonymous non-terminals are introduced, the reduction of which entails no semantic actions. The burden of constructing a list of the recognized *A*s lies entirely on the semantic routines attached to the *A*s.

The iterative interpretation causes more problems. When an A^+ has been recognized and is about to be reduced, the stack holds an indeterminate number of *A*s:

$$\cdots A \cdots AAA \quad |$$

The right end of the handle has been found, but the left end is doubtful. Scooping up all *A*s from the right may be incorrect since some may belong to another rule; after all, the top of the stack may derive from a rule $P \rightarrow QAAA^+$. A possible solution is to have for each reducing state and look-ahead a FS automaton which scans the stack backwards while examining states in the stack to determine the left end and the actual rule to reduce to. The part to be reduced (the handle) can then be shown to a semantic routine which can, for example, construct a list of *A*s, thereby relieving the *A*s from a task that is not structurally theirs. The resulting tables can be enormous and clever algorithms have been designed for their construction and reduction. See for example, LaLonde [62], Nakata and Sassa [69, 74], Shin and Choe [90], Fortes Gálvez, [91], and Morimoto and Sassa [97]. Kannapinn [99] has given a critical analysis of many algorithms for LR and LALR parse table creation for EBNF grammars (in German).

9.10.4 Incremental Parsing

In incremental parsing, the structured input (a program text, a structured document, etc.) is kept in linear form together with a parse tree. When the input is (incrementally) modified by the user, for example by typing or deleting a character, it is the task of the incremental parser to update the corresponding parse tree, preferably at minimum cost. This requires serious measures inside the parser, to quickly determine the extent of the damage done to the parse tree, localize its effect, and take remedial steps. Formal requirements for the grammar to make this easier have been found. See for example, Degano, Mannucci and Mojana [330] and many others in (Web)Section 18.2.8.

9.10.5 Incremental Parser Generation

In incremental parser generation, the parser generator keeps the grammar together with its parsing table(s) and has to respond quickly to user-made changes in the grammar, by updating and checking the tables. See Horspool [80], Heering, Klint and Rekers [83], Horspool [84] and Rekers [347].

9.10.6 Recursive Ascent

In Sections 8.2.6 and 8.5 we have seen that an LL parser can be implemented conveniently using recursive descent. Analogously, an LR parser can be implemented using *recursive ascent*, but the required technique is not nearly as obvious as in the LL case. The key idea is to have the recursion stack mimic the LR parsing stack. To this end there is a procedure for each state; when a token is to be shifted to the stack, the procedure corresponding to the resulting state is called instead. This indeed constructs the correct recursion stack, but causes problems when a reduction has to take place: a dynamically determined number of procedures has to return in order to unstack the right-hand side. A simple technique to achieve this is to have two global variables, one, *Nt*, holding the non-terminal recognized and the second, *l*, holding the length of the right-hand side. All procedures will check *l* and if it is non-zero, they will decrease *l* by one and return immediately. Once *l* is zero, the procedure that finds that situation will call the appropriate state procedure based on *Nt*. For details see Roberts [78, 81, 87] and Kruseman Aretz [77]. The advantage of recursive ascent over table-driven is its potential for high-speed parsing.

9.10.7 Regular Expressions of LR Languages

In Section 9.6.3 we mentioned that any LR(k) language can be obtained as a regular expression, the elements of which are LR(0) languages. The opposite is even stronger: regular expressions over LR(0) languages can describe more than the LR(k) languages. An immediate example is the inherently ambiguous language $\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n \cup \mathtt{a}^p\mathtt{b}^p\mathtt{c}^q$ discussed on page 64. It is produced by the regular expression

$$\mathcal{L}_{\mathtt{a}}^*\mathcal{L}_{\mathtt{bc}}|\mathcal{L}_{\mathtt{ab}}\mathcal{L}_{\mathtt{c}}^*$$

where the language $\mathcal{L}_{\mathtt{a}}$ is produced by the simplest grammar in this book, `S->a`, $\mathcal{L}_{\mathtt{bc}}$ by `S->bSc`$|\varepsilon$, and similarly for $\mathcal{L}_{\mathtt{ab}}$ and $\mathcal{L}_{\mathtt{c}}$. It is easy to see that each of these grammars is LR(0).

Bertsch and Nederhof [96] show that a linear-time parser can be constructed for regular expressions over LR(k) languages. Unfortunately the algorithm is based on descriptions of the languages by pushdown automata rather than CF grammars, and a transformation back to CF grammars would be very complicated. Some details are provided in Section 12.3.3.2, where a similar technique is used for linear-time substring parsing of LR languages.

9.11 Getting a Parse Tree Grammar from LR Parsing

Getting a parse tree grammar from LR parsing is similar to getting one from LL parsing (Section 8.4): each time one makes a "serious" decision (prediction, reduction) one generates a grammar rule for it. As in the LL case, LR parsing produces a parse tree grammar rather than a parse forest grammar.

We consider parsing **n-n** with the table of Figure 9.18. All non-terminals are numbered using the same increasing counter. After a single shift we have the configuration

① **n** ③ **- n $**

The moment we reduce **n** to **T**, we produce a rule **T_1→n**, and push **T_1** on the stack:

① **T_1** ② **- n $**

The next step reduces the **T** to **E**; this produces a rule **E_2→T_1** and the configuration

① **E_2** ② **- n $**

Continuing this process we obtain the parse tree grammar

```
T_1  →  n
E_2  →  T_1
T_3  →  n
E_4  →  E_2 - T_3
S_5  →  E_4 $
```

and the final datum yielded by the parsing process is that **S_5** is the start symbol of the parse tree grammar.

Note that it is possible to number the input tokens with their positions and to follow where they go in the parse tree grammar:

```
T_1  →  n1
E_2  →  T_1
T_3  →  n3
E_4  →  E_2 -2 T_3
S_5  →  E_4 $4
```

This is useful when semantics is attached to the input tokens.

Again the grammar is clean. It has no undefined non-terminals: each non-terminal included in a right-hand side during a reduction comes from the stack, and was defined in a previous reduction. It has no unreachable non-terminals either: each left-hand side non-terminal created in a reduction is put on the stack, and will later be included in some right-hand side during a subsequent reduction, except for the start symbol, which is reachable by definition.

9.12 Left and Right Contexts of Parsing Decisions

At the beginning of Chapter 7 we indicated that stacks in bottom-up parsing can be described by regular grammars, and we are now in a position to examine this phenomenon in more detail, by considering two non-obvious properties of an LR automaton: the left context of a state and the right context of an item.

9.12.1 The Left Context of a State

The left context of a state is easy to understand: it is the set of all sequences of symbols, terminals and non-terminals, that lead to that state. Although this set is usually infinitely large, it can be represented by a regular expression. It is easy to see that, for example, the left context of state 4 in the LR automaton in Figure 9.17 is **E**, but more work is needed to obtain the left context of, say, state 9. To find all paths that end in state 9 we proceed as follows. We can create the path to state 9 if we know the path(s) to state 6 and then append an **E**. This gives us one rule in a left-regular grammar: **P_9→P_6 E**, where **P_6** and **P_9** are the paths to states 6 and 9, respectively. Now there are three ways to get to state 6: from 1, from 6 and from 7, all through a **(**. This gives us three rules: **P_6→P_1(**, **P_6→P_6(**, **P_6→P_7(**. Continuing in this way we can construct the entire left-context grammar of the LR automaton in Figure 9.17. It is shown in Figure 9.43, and we see that it is left-regular.

```
P_1 → ε       P_4 → P_1 E   P_7 → P_4 -
P_2 → P_1 T   P_5 → P_4 $   P_7 → P_9 -
P_2 → P_6 T   P_6 → P_1 (   P_8 → P_7 T
P_3 → P_1 n   P_6 → P_6 (   P_9 → P_6 E
P_3 → P_6 n   P_6 → P_7 (   P_10 → P_9 )
P_3 → P_7 n
```

Fig. 9.43. Left-context grammar for the LR(0) automaton in Figure 9.17

We can now apply the transformations shown in Section 5.4.2 and Section 5.6 to obtain regular expressions for the non-terminals. This way we find that indeed the left context of state 4 is **E** and that that of state 9 is **[(|E-()[(|E-()*****E**. This expression simplifies to **[(|E-()**$^+$**E**, which makes sense: it describes a sequence of one or more **(** or **E-(**, followed by an **E**. The first **(** or **E-(** brings us to state 6, any subsequent **(**s and **E-(**s bring us back to state 6, and the final **E** brings us to state 9.

Now the connection with the stack in an LR parser becomes clear. Such a stack can only consist of a sequence which leads to a state in the LR automaton; for example, it could not be **(-**, since that leads nowhere in Figure 9.17, though it could be **(E-** (which leads to state 7). In short, the union of all left contexts of all states describes the complete set of stack configurations of the LR parser.

All stack configurations in a given *P_s* end in state *s* and thus lead to the same parsing decision. LR(1) automata have more states than LR(0) automata, and thus more left context sets. For example, the LR(1) automaton in Figure 9.27 remembers whether it is working on the outermost expression (in which case a **#** may follow) or on a nested expression; the LR(0) automaton in Figure 9.17 does not. But the set of all stack configurations *P_*$*$ is the same for LR(0) and LR(1), because they represent all open parts in a rightmost production, as explained in Section 5.1.1.

9.12.2 The Right Context of an Item

The right context of a state is less easy to understand: intuitively it is the set of all strings that are acceptable to an LR parser in that state, but that set differs considerably from the left context sketched above.

First it is a context-free language rather than a regular one. This is easy to see when we consider an LR parser for a CF language: any string in that language is acceptable as the right context of the initial state.

Second, it contains terminals only; there are no non-terminals in the rest of the input, to the right of the gap. Yet it is clear that the right context of an item is not just an unrestricted set of strings, but follows precisely from the CF grammar C and the state S, and we would like to capture these restrictions in a grammar. This is achieved by constructing a regular grammar G_S for the right context which still contains undeveloped non-terminals from C, similar to the left context grammar. The set of terminal strings acceptable after a state is then obtained by replacing these non-terminals by their terminal productions in C; this introduces the CF component. More precisely: each (regular) terminal production T_S of the grammar G_S is a start sentential form for grammar C; each combination of T_S and C produces a (CF) set of strings that can figure as rest of input at S.

There is another, perhaps more surprising, difference between left and right contexts: although the left contexts of all items in an LR state are the same, their right contexts can differ. The reason is that the same LR state can be reached by quite different paths through the grammar. Each such path can result in a different item in that state and can carry a different prediction of what will happen on the other side of the item. A trivial example occurs in the grammar

```
Ss  →  a B c
 S  →  a D e
 B  →  ε
 D  →  ε
```

The state reached after shifting over an **a** contains

```
S→a•Bc
S→a•De
B→•
D→•
```

and it is clear that the right context of the item **B→•** is **c** and that of **D→•** is **e**. This example already alerts us to the relationship between right contexts and lookahead symbols. Like the latter (Section 9.6.2) right contexts exist in an item and a dot variety. The item right context of **S→a•Bc** is ε; its dot right context is **Bc**. Item right contexts are easier to compute but dot right contexts are more important in parsing.

We shall start by constructing the regular grammar for item right contexts for the automaton in Figure 9.17, and then derive dot right contexts from it. Since the right contexts are item-specific we include the names of the items in the names of the non-terminals that describe them. We use names of the form **F_***s*{*I*} for the set of strings that can follow item I in state s in sentential forms during rightmost derivation.

As we have seen in Section 9.5, items can derive from items in the same state or from a parent state. An example of the first type is `E->•E-T` in state 6. It derives through the ε-moves `T->(•E)` $\xrightarrow{\varepsilon}$ `•E` $\xrightarrow{\varepsilon}$ `E->•E-T` and `E->•E-T` $\xrightarrow{\varepsilon}$ `•E` $\xrightarrow{\varepsilon}$ `E->•E-T` in the non-deterministic automaton of Figure 9.15 from both `T->(•E)` and `E->•E-T` in state 6. An example of the second type is `T->(•E)` in state 6, deriving in three ways from the `T->•(E)` in states 1, 6 and 7, through the transition `T->•(E)` $\xrightarrow{(}$ `T->(•E)` in Figure 9.15.

If the item `E->•E-T` originates from `T->(•E)`, its right context consists of the `)` which follows the `E` in `T->(•E)`; this gives one rule for `F_6{E->•E-T}`:

```
F_6{E->•E-T} -> ) F_6{T->(•E)}
```

If the item originates from `T->•E-T`, its right context consists of the `-T` which follows the `E` in `T->•E-T`; this gives the second rule for `F_6{E->•E-T}`:

```
F_6{E->•E-T} -> -T F_6{E->•E-T}
```

The general rule is: `F_`s{$A \to \bullet\alpha$} `->` γ `F_`s{$X \to \beta\bullet A\gamma$} for an ε-transition $\{X \to \beta\bullet A\gamma\} \xrightarrow{\varepsilon} \{\bullet A\} \xrightarrow{\varepsilon} \{A \to \bullet\alpha\}$, for each state s in which the item $\{X \to \beta\bullet A\gamma\}$ occurs.

A shift over a token does not change the right context of an item: during a shift over a `(` from state 1 to state 6, the item `T->•(E)` changes into `T->(•E)`, but its right context remains unaffected. This is expressed in the rule

```
F_6{T->(•E)} -> F_1{T->•(E)}
```

The general rule is: `F_`r{$A \to \alpha t\bullet\beta$} `->` `F_`$s${$A \to \alpha\bullet t\beta$} for a transition $\{A \to \alpha\bullet t\beta\} \xrightarrow{t} \{A \to \alpha t\bullet\beta\}$.

Repeating this procedure for all ε-moves and shifts in Figure 9.17 gives us the (right-regular) grammar for the right contexts; it is shown in Figure 9.44. Note that the two ways of propagating right context correspond with the two ways of propagating the one-token look-ahead in LR(1) parsing, as explained on page 293.

Again applying the transformations from Section 5.4.2 we can obtain regular expressions for the non-terminals. For example, the item right context of `E->•E-T` in state 6 is `[)`*`[-T|)]]`*`)`*`)[-T]`*`$` which simplifies to `[-T|)]`*`)[-T]`*`$`. Again this makes sense: the prediction after that item is a sequence of `-T`s and `)`s, with at least one `)`, since to arrive at state 6, the input had to contain at least one `(`.

Finding dot right contexts is now simple: the dot right context of `E->•E-T` in state 6, `D_6{E->•E-T}`, is of course just `E-T F_6{E->•E-T}`. The general rule is: `D_`s{$A \to \alpha\bullet\beta$} `->` β `F_`s{$A \to \alpha\bullet\beta$} for all items.

For a thorough and formal analysis of right contexts see Seyfarth and Bermudez [93].

9.13 Exploiting the Left and Right Contexts

There are many ways to exploit the left and right contexts as determined above. We will discuss here three techniques. The first, DR(k) parsing, uses knowledge of the

```
F_1{S→•E$} → ε
F_1{E→•E-T} → -T F_1{E→•E-T}
F_1{E→•E-T} → $ F_1{S→•E$}
F_1{E→•T} → -T F_1{E→•E-T}
F_1{E→•T} → $ F_1{S→•E$}
F_1{T→•(E)} → F_1{E→•T}
F_1{T→•n} → F_1{E→•T}
F_2{E→T•} → F_1{E→•T}
F_2{E→T•} → F_6{E→•T}
F_3{T→n•} → F_1{T→•n}
F_3{T→n•} → F_6{T→•n}
F_3{T→n•} → F_7{T→•n}
F_4{S→E•$} → F_1{S→•E$}
F_4{E→E•-T} → F_1{E→•E-T}
F_5{S→E$•} → F_4{S→E•$}
F_6{E→•E-T} → ) F_6{T→(•E)}
F_6{E→•E-T} → -T F_6{E→•E-T}
F_6{E→•T} → -T F_6{E→•E-T}
F_6{E→•T} → ) F_6{T→(•E)}
F_6{T→(•E)} → F_1{T→•(E)}
F_6{T→(•E)} → F_6{T→•(E)}
F_6{T→(•E)} → F_7{T→•(E)}
F_6{T→•(E)} → F_6{E→•T}
F_6{T→•n} → F_6{E→•T}
F_7{E→E-•T} → F_4{E→E•-T}
F_7{E→E-•T} → F_9{E→E•-T}
F_7{T→•(E)} → F_7{E→E-•T}
F_7{T→•n} → F_7{E→E-•T}
F_8{E→E-T•} → F_7{E→E-•T}
F_9{E→E•-T} → F_6{E→•E-T}
F_9{T→(E•)} → F_6{T→(•E)}
F_10{T→(E)•} → F_9{T→(E•)}
```

Fig. 9.44. Right-regular right-context grammar for the LR(0) automaton in Figure 9.17

left context to reduce the required table size drastically, while preserving full LR(k) parsing power. The second, LR-regular, uses the full right context to provide optimal parsing power, but the technique does not lead to an algorithm, and its implementation requires heuristics and/or handwaving. The third, LAR(k) parsing, is a tamed version of LR-regular, which yields good parsers for a large class of unambiguous grammars. An even more extensive application of the contexts is found in the chapter on non-canonical parsing, Chapter 10, where the right context is explicitly improved by doing reductions in it. And there is no reason to assume that this exhausts the possibilities.

9.13.1 Discriminating-Reverse (DR) Parsing

As Figure 9.12 shows, an LR parser keeps states alternatingly between the stacked symbols. Actually, this is an optimization; we could omit the states, but that would force us to rescan the stack after each parse action, to reestablish the top state, which would be inefficient. Or would it? Consider the sample parser configuration on page 283, which was based on the grammar of Figure 9.14 and the handle recognizer of Figure 9.17, and which we repeat here without the states:

```
# E - n ❍                    - n $
```

We have also added a bottom-of-stack marker, **#**, which can be thought of as caused by stacking the beginning of the input. There can be no confusion with the end-of-input marker **#**, since the latter will never be put on the stack.

When we look at the above configuration, we find it quite easy to guess what the top state, indicated by ❍, must be. The last shift was over an **n**, and although there are three arrows marked **n** in Figure 9.17, they all point to the same state, ③, so we can be certain we are looking at

```
# E - n ③                    - n $
```

Since ③ is a reduce state, we can confidently reduce the **n** to **T**. So the loss of the left context did not do any harm here; but, as the reader might expect, that is not going to last.

After the reduction we have the configuration

```
# E - T ❍                    - n $
```

and now we have a problem. There are again three arrows marked **T** in Figure 9.17, but they do not all point to the same state; two point to ② and one points to ⑧, so we seem none the wiser. But we know how to handle situations in which there are only a finite number of possibilities: we put them all in a state, and progress with that state as our knowledge. The state is

```
①: ②
⑥: ②
⑦: ⑧
```

and it represents our knowledge between the **-** and the **T** in the configuration; it says: if we are now in LR state ① or ⑥, the top state was ②, and if we are now in state ⑦, the top state was ⑧. Such a state is called a *DR* state, for *Discriminating Reverse* state (Fortes Gálvez [89]).

When we now look backwards on the stack, we see a **-**; the LR states ① and ⑥ do not have incoming arrows marked **-**, so we cannot be in one of these, but state ⑦ has, coming from ④ and ⑨. So our DR state between the **E** and the **-** is

```
④: ⑧
⑨: ⑧
```

which says if we are now in LR state ④, the top state was ⑧, and if we are now in state ⑨, the top state was ⑧. Here something surprising has happened: even though we do not know in which LR state we are, we now know that the top of the stack was ⑧, which is the answer we were looking for! This gives us the configuration

```
# E - T ⑧                    - n $
```

in which we reduced the **E-T** to **E**. We have now reproduced the parsing example of page 283 without states on the stack and without rescanning the entire stack for each parsing action. Whether that is something worth striving for is less than clear for the moment, but we will see that the technique has other benefits.

More generally, suppose we are in a DR state d

$$
\begin{array}{l}
l_1\texttt{:}\ t_1 \\
\cdots \\
l_k\texttt{:}\ t_k
\end{array}
$$

where $l_1 \cdots l_k$ are LR states and the $t_1 \cdots t_k$ are the top-of-stack states implied by them, and suppose we have the stack symbol s on our left. Now we want to compute the DR state to the left of s, one step back on the stack. To do so we go through all transitions of the form $p_1 \xrightarrow{s} p_2$ in the LR handle recognizer, and for each transition

that has an p_2 equal to an l_j, we insert the form $p_1 : t_j$ in a new DR state e. This is reasonable because if we were in LR state p_1 to the left of the s, then moving over the s would bring us in LR state p_2, and that would imply that the top of the stack is t_j. In this way we obtain a transition in a *DR automaton*: $d \xrightarrow{s} e$, or more graphically, $e \xleftarrow{s} d$. This transition carries our knowledge about the top of the stack in our present position over the symbol s to the left.

We can compute the complete DR automaton by performing this step for all possible stack symbols, starting from the initial state of the DR automaton

$t_1 : t_1$
$\cdots$
$t_k : t_k$

which of course says that if we are in LR state t_j on the top of the stack, then the top-of-stack state is t_j. It is always good to see a difficult concept reduced to a triviality. States in which all $t_1 \cdots t_k$ are equal are final states, since they unequivocally tell us the top-of-stack state. The DR automaton generation process is guaranteed to terminate because there are only a finite number of DR states possible. The DR automaton for the LR automaton of Figure 9.17 is shown in Figure 9.45.

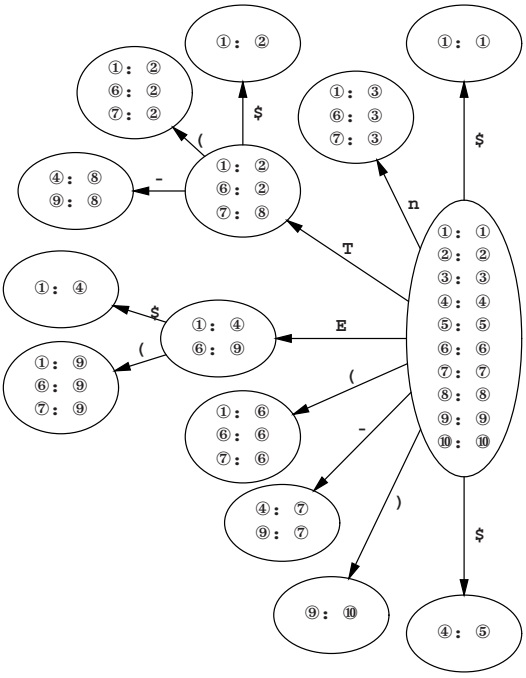


Fig. 9.45. DR automaton for the LR automaton of Figure 9.17

One thing we immediately notice when looking at the graph in Figure 9.45 is that it has no loops: at most two steps backwards suffice to find out which parsing action is called for. We have just shown that the grammar of Figure 9.14 is BRC(2,0)!

But there are more important things to notice: now that we have the transition diagram in Figure 9.45 we can discard the GOTO table of the LR parser (but of course we have to keep the ACTION table). That looks like a meager advantage: the DR automaton has 14 states and the LR(0) automaton only 10. But DR automata have an interesting property, already showing up in Figure 9.45: the first fan-out is equal to the number of symbols, the second fan-out is usually a modest number, the third fan-out a very modest number and in many DR tables there is no fourth fan-out. This is understandable, since each step to the left tends to reduce the uncertainty. Of course it is possible that some DR parser will occasionally dig unboundedly deep back in the stack, but such operations are usually controlled by a simple loop in the DR automaton (see Problem 9.21), involving only a few DR states.

Compared to GOTO tables the DR automata are very compact, and, even better, that property holds more or less independently of the type of LR table used: going from LR(0) to LR(1) to LR(2) tables, each one or more orders of magnitude larger than the previous, the corresponding DR automaton only grows minimally. So we can afford to use full LR(k) tables and still get a very small replacement for the GOTO table! We still need to worry a bit about the ACTION table, but almost all of its entries are "shift" or "error", and it yields readily to table compression techniques. DR parsing has been used to create LR(1) parsers that are substantially smaller than the corresponding LALR(1) tables. The price paid for these smaller tables is an increased parse time caused by the stack scanning, but the increase is very moderate.

The reader may have noticed that we have swept two problems under the rug in the above explanation: we needed the large LR table to obtain the small DR table, a problem similar to the construction of LALR(1) parsers without generating the full LR(1) tables; and we ignored look-aheads. Solving these problems is the mainstay of DR parser generation; detailed solutions are described by Fortes Gálvez [92, 95]. The author also proves that parse time is linear in the length of the input, even if the parser sometimes has to scan the entire stack [95, Section 7.5.1], but the proof is daunting. A generalized version of DR parsing is reported by Fortes Gálvez et al. [179] and a non-canonical version by Farré and Fortes Gálvez [207, 209].

Kannapinn [99] describes a similar system, which produces even more compact parsers by first reducing the information contents of the LR(1) parser, using various techniques. This reduction is, however, at the expense of the expressive power, and for stronger reductions the technique produces smaller parsers but can handle fewer grammars, thus defining a number of subclasses of LR(1).

9.13.2 LR-Regular

The right context can be viewed as a kind of super-look-ahead, which suggests that it could be a great help in resolving inadequate states; but it is not particularly easy to make this plan work. The basic idea is simple enough: whenever we meet an inadequate state in the parse table construction process, compute the right contexts of

the offending items as described in the previous section. If the two contexts describe disjunct sets, they can serve to resolve the conflict at parse time by finding out to which of the two sets the rest of the input belongs. If the two contexts do not exclude each other, the plan does not work for the given grammar. (See Figure 9.13 for a simple unambiguous grammar for which this technique clearly will not work.)

This requires us to solve two problems: deciding whether the two dot right contexts are disjunct, and checking the rest of the input against both contexts. Both are serious problems, since the right contexts are CF languages. It can be proved that it is undecidable whether two CF languages have a terminal production in common, so finding out if the two right contexts really are sufficient to distinguish between the two items seems impossible (but see Problem 9.29). And checking the rest of the input against a CF language amounts to parsing it, the very problem we are trying to solve.

Both problems are solved by the same trick: we replace the CF grammars of the right contexts by regular grammars. As we have seen in Section 5.5 we can check if two regular languages are disjunct (take the intersection of the automata of both languages and see if the resulting automaton still accepts some string; if it does, the automata are not disjunct). And it is simple to test the rest of the input against both regular languages; below we will show that we can even do that efficiently. But this solution brings in a new problem: how to replace CF grammars by regular ones.

Of course a regular grammar R cannot be equivalent to a CF grammar C, so replacing one by the other involves an approximation "from above": R should at least produce all strings C produces or it will fail to identify an item as applicable when it is. But the overproduction should be minimal, or the set of string may no longer be disjunct from that of the other item, and the parser construction would fail unnecessarily. So R will have to *envelop* C as tightly as possible. If mutually disjunct regular envelopes for all right contexts in inadequate states exist, the grammar G is *LR-regular* (Čulik, II and Cohen [57]), but we can make a parser for G only if we can also actually find the envelopes.

It is actually not necessary to find regular envelopes of the right contexts of each of the items in an inadequate state. It suffices to find regular envelopes for the non-terminals of the grammar; these can then be substituted into the regular expressions for the right contexts.

Finding regular envelopes of non-terminals in a context-free grammar requires heuristics. It is possible to approximate non-terminals better and better with increasingly more complicated regular grammars, but it is undecidable if there exist regular envelopes for the right contexts that are good enough for a given grammar. So when we find that our approximations (regular envelopes) are not disjunct, we cannot know if better heuristics would help. We shall therefore restrict ourselves to the simple heuristic demonstrated in Section 9.13.2.3.

9.13.2.1 LR-Regular Parse Tables

Consider the grammar in Figure 9.46(*a*), which produces `d`*`a` and `d`*`b`. It could, for example, represent a language of integer numbers, with the `d`s standing for digits,

```
Ss  →  A a        S→•Aa
S   →  B b        S→•Bb
A   →  A C        A→•AC
A   →  C          A→•C        C→d•
C   →  d          C→•d        D→d•
B   →  B D        B→•BD         (c)
B   →  D          B→•D
D   →  d          D→•d
    (a)             (b)
```

Fig. 9.46. An LR-regular grammar (*a*), with initial state 1 (*b*) and inadequate state 2 (c)

and the **a** and **b** for indications of the numeric base; examples could then be **123a** for a decimal number, and **123b** for a hexadecimal one. For another motivation of this grammar see Section 10.2.2 and Figure 10.12.

It is easy to see that the grammar of Figure 9.46(*a*) is not LR(*k*): to get past the first **d**, it has to be reduced to either **C** or **D**, but no fixed amount of look-ahead can reach the deciding **a** of **b** at the end of the input. The figure also shows the initial state 1 of an LR parser for the grammar, and the state reached by shifting over a **d**, the one that has the reduce/reduce conflict. The full LR automaton is shown in Figure 9.49.

To resolve that conflict we construct the right contexts of both items, **F_2{C→d•}** and **F_2{D→d•}**. The regular grammar for **F_2{C→d•}** is

```
F_1{S→•Aa}  →  #
F_1{A→•AC}  →  a F_1{S→•Aa}
 F_1{A→•C}  →  a F_1{S→•Aa}
F_1{A→•AC}  →  C F_1{A→•AC}
 F_1{A→•C}  →  C F_1{A→•AC}
 F_1{C→•d}  →  F_1{A→•C}
 F_2{C→d•}  →  F_1{C→•d}
```

Unsurprisingly this resolves into **C*a#**. A similar reasoning gives **D*b#** for **F_2{D→d•}**. Next we have to replace the CF non-terminals **C** and **D** by their regular envelopes. In our example this is trivial, since both are already regular; so the two LR-regular contexts are **d*a#** and **d*b#**. And indeed the two sets are disjunct: the grammar of Figure 9.46(*a*) is LR-regular, and the LR-regular contexts can be used as LR-regular look-aheads. Right contexts always end in a **#** symbol, since each item eventually derives from the start symbol, and it has a look-ahead **#**.

So the entry for a state *p* in the ACTION table of an LR-regular parser can contain one of five things: “shift”, “reduce”, “error”, “accept”, or a pointer to an LR-regular look-ahead automaton; the latter occurs when the LR state corresponding to *p* was inadequate. The GOTO table of an LR-regular parser in identical to that of the LR parser it derives from.

9.13.2.2 Efficient LR-Regular Parsing

We now turn to the actual parsing, where we meet our second problem: how to determine which of the right contexts the rest of the input is in. The naive way is to just construct an FSA for each LR-regular look-ahead and send it off into the input to see if it stops in an accepting state. This has two drawbacks: 1. the input is rescanned by each FSA F, and there can be many of them; 2. the whole process is repeated after each shift, which may cause the parsing to require $O(n^2)$ time.

The second drawback can be removed by replacing the FSA F by a new FSA $\overleftarrow{F}$, which accepts the reverse of the strings that F accepts; basically such an FSA can be made by reversing all arrows, swapping the initial and accepting states, and making the result deterministic again. We start $\overleftarrow{F}$ at the right of the added end-of input token `#`, and run it backwards over the input. It marks each position in which it is in an accepting state with a marker F_1, the start state of the original, forward, automaton F. This costs $O(n)$ steps. Now, when during parsing we want to know if the rest of the input conforms to F, we can just check if the present position is marked F_1, at constant cost.

We can of course repeat the backward scan of the input for every reversed look-ahead FSA, but it is much more efficient to combine all of them in one big FSA $\overleftarrow{\mathcal{F}}$ by creating a new start state ⊛ with ε-transitions to the start states of all reversed automata for the dot right contexts, as shown in Figure 9.47. The clouds represent

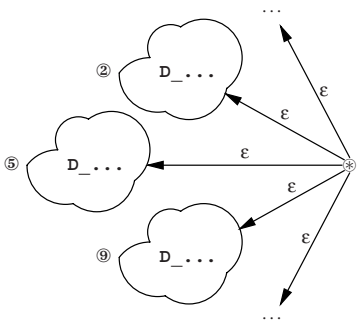
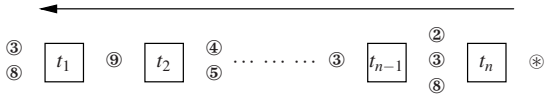


Fig. 9.47. Combined backwards-scanning automaton $\overleftarrow{\mathcal{F}}$ for LR-regular parsing

the various reversed automata, with their accepting states ②, ⑤, ⑨, etc. Using this combined automaton we need now scan backwards only once:



The backwards scan marks each position with the accepting states of all reversed FSAs in $\overleftarrow{\mathcal{F}}$ that apply at that position. These are the start states of the forward automata. A left-to-right LR parsing scan can then use these states as summaries of the look-aheads. This removes the first drawback mentioned above.

We have now achieved a linear-time algorithm: we first read the entire input (at cost $O(n)$); then we scan backwards, using one single FSA recording start states of right contexts (again $O(n)$); and finally we run the LR-regular parser forward, using the recorded states rather than the tokens as look-aheads (also $O(n)$).

9.13.2.3 Finding a Regular Envelope of a Context-Free Grammar

The fundamental difference between regular and context-free is the ability to nest. This nesting is implemented using a stack, both during production and parsing, for LL, LR and pushdown automaton alike. This observation immediately leads to a heuristic for "reducing" a CF language to regular: limit the stack depth. A stack of fixed depth can assume only a finite number of values, which then correspond to the states of a finite state automaton. The idea can be applied naturally to an LR parser with a stack limited to the top state only (but several other variations are possible).

The heuristic can best be explained using a non-deterministic LR automaton, for example the one in Figure 9.15. Assume the input is `n($`. Initially we work the system as an interpreter of the NFA, as in Figure 9.16, so we start in the leftmost state in that figure. Shifting over the `n` brings us to a state that contains only `T→n•` (actually state 2 in Figure 9.17), and since we remember only the top of the stack, we forget the initial state. State 2 orders us to reduce, but since we have lost the `n` *and* the initial state, we know that we have to shift over a `T` but we have no idea from what state to shift. We solve this by introducing a state containing all possible items, thus acknowledging our total ignorance; we then shift over `T` from that state. The result is the item set:

```
E→E-T•
E→T•
```

Note that this item set is not present in the deterministic LR(0) automaton, and cannot occur as a state in the CF parsing. The item set tells us to reduce to `E`, but again without any previous information. We act as above, now obtaining the item set

```
S→E•$
E→E•-T
T→(E•)
```

which is again not an LR(0) state. This item set allows shifts on `$`, `-` and `)`, but not on `(`; so the input `n($` is rejected, even by the regular envelope constructed here. Note that the input `n)$` *is* accepted; indeed it does not contain blatant impossibilities.

A closer look at the above discussion makes it clear what happens when we have to reduce to a non-terminal A: we continue with all items of the form $P \rightarrow \alpha A \bullet \beta$. These items can be found in the non-deterministic LR automaton as the items that have an incoming arrow on A. This gives us a way to convert such an automaton

into an FSA for a regular envelope: we connect by ε-transitions all reduce states for each non-terminal A to all states with incoming arrows marked A; next we remove all arrows marked with non-terminals.

This procedure converts the non-deterministic LR(0) automaton of Figure 9.15 into the non-deterministic finite-state automaton of Figure 9.48, in which the unmarked arrows represent ε-transitions, and the accepting state is again marked with a ◊. Rather than connecting all reduce items of a non-terminal A to all items of the

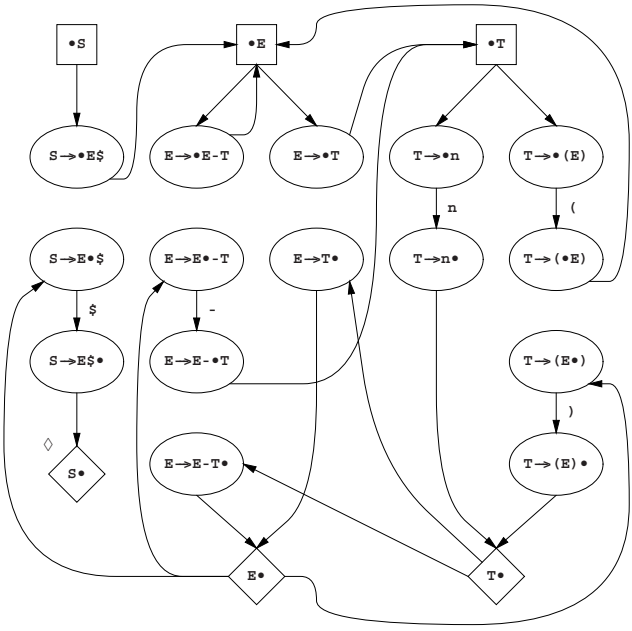


Fig. 9.48. A possible regular envelope for the grammar of Figure 9.14

form $P \rightarrow \alpha A \bullet \beta$, we first connect the reduce items to a "terminal station", which is the dual to the "departure" station shown in Figure 9.15, and connect from there to the destination states. Although Figure 9.48 could be drawn neater and without crossing lines, we have kept it as close as possible to Figure 9.15 to show the relationship.

A specific deterministic finite-state automaton for a given non-terminal P can be derived from it by marking the station of P as the start state, and making the automaton deterministic using the subset algorithm. This FSA — or rather the regular expression it corresponds to — can then be used in the expressions derived for item and dot right contexts in Section 9.12.2. See Problem 9.27.

If the resulting regular sets are too coarse and do not sufficiently separate the actions on various items, a better approximation could be obtained by remembering k states rather than 1, but the algorithm to do so is quite complicated. It is usually much easier to duplicate part of the grammar, for example as follows:

```
 S  →  E $
 E  →  E - T' | T
 T  →  n | ( E )
T'  →  n | ( E )
```

This trick increases the number of states in the FSA and so the tightness of the fit. But finding exactly which part to duplicate will always remain an art, since the basic problem is unsolvable.

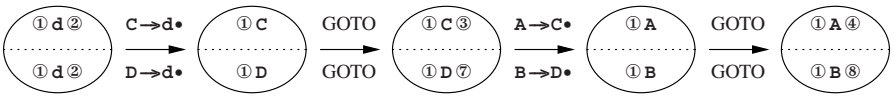
The grammar of Figure 9.46(*a*) shows that LR-regular parsing can handle some non-deterministic grammars. Čulik, II and Cohen [57] prove that the same is true for languages: LR-regular can handle some languages for which there are no deterministic grammars. For the dismal error detection properties of LR-regular, see Problem 9.28.

The above approximation algorithm is from Nederhof [402]. There are many other algorithms for approximating the right context, for example Farré and Fortes Gálvez [98]. See also Yli-Jyrä [403], Pereira and Wright [404], and other papers from (Web)Section 18.4.2. Nederhof's paper [401] includes a survey of regular approximating algorithms.

9.13.3 LAR(*m*) Parsing

Bermudez and Schimpf [82] show a rather different way of exploring and exploiting the right context. At first sight their method seems less than promising: when faced with two possible decisions in an inadequate state, parse ahead with both options and see which one survives. But it is easy to show that, at least occasionally, the method works quite well.

We apply the idea to the grammar of Figure 9.46. Its LR(0) automaton is shown in Figure 9.49; indeed state ② is inadequate, has a reduce/reduce conflict. Suppose the input is **dddb**, which almost immediately lands us in the inadequate state. Rather than first trying the reduction **C→d•** and seeing where it gets us, and then **D→d•**, we try both of them simultaneously, one step at a time. In both cases the parser starts in state ①, a **d** is stacked, and state ② is stacked on top, as in frame *a*:



The top level in the bubble is reduced using **C→d•** and the bottom level with **D→d•**, as shown in frame *b*. GOTOs over the resulting **C** and **D** give frame *c*. The new states ③ and ⑦ are OK and ask for more reductions, leading to frame *d*. Two more GOTOs put the states ④ and ⑧ on top; both require shifts, so our simultaneous parser is now ready for the next input token. The way we have drawn the combined simulated stacks in transition bubbles already shows that we intend to use them as states in a look-ahead automaton, the *LAR automaton*.

When we now process the next **d** in the input:

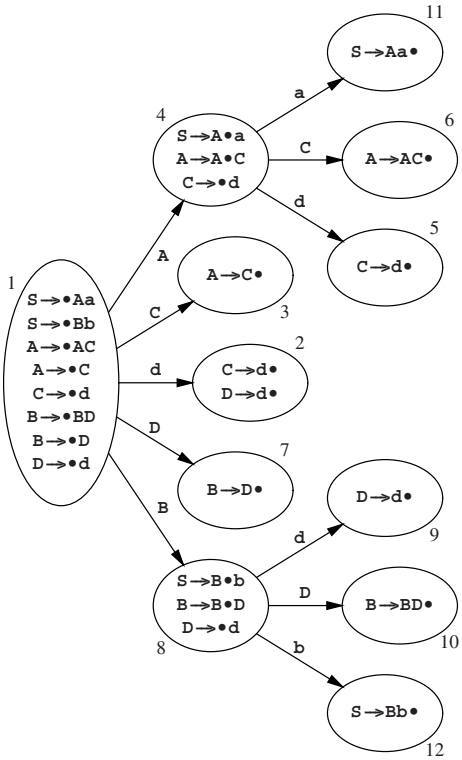
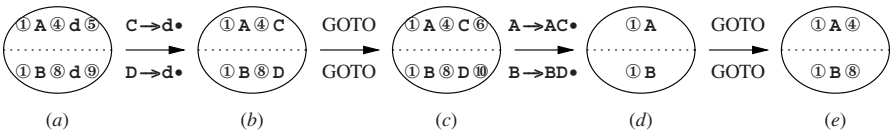
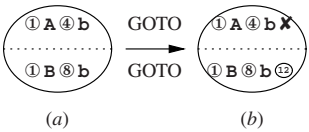


Fig. 9.49. The LR(0) automaton of the grammar of Figure 9.46



we are pleasantly surprised: the LAR state after the second **d** is the same as after the first one! This means that any further number of **d**s will just bring us back to this same state; we can skip explaining these and immediately proceed to the final **b**. We stack the **b** and immediately see that one of the GOTOs fails ((*b*)):



That is all we need to know: as soon as there is only one choice left we can stop our search since we know which decision to take in the inadequate state.

It would be inconvenient to repeat this simulation every time the inadequate state occurs during parsing, so we want to derive from it a finite-state look-ahead automaton that can be computed during parser generation time and can be consulted during parsing. To this end we perform the simulated look-ahead process during parser generation, for all input tokens. This results in a complete FS look-ahead automaton for the given inadequate state. Figure 9.50 shows the LAR automaton for the inadequate state ②, as derived above. Note that it is exactly the FS automaton a programmer

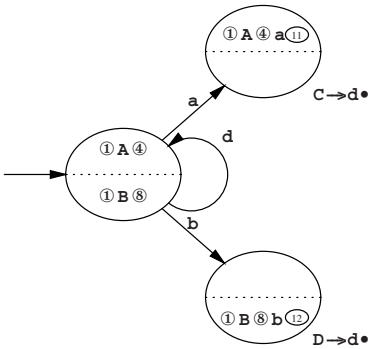


Fig. 9.50. The LAR automaton for the inadequate state ② in Figure 9.49

would have written for the problem: skip **d**s until you find the answer.

The above example allowed us to demonstrate the basic principles and the power of LAR parsing, but not its fine points, of which there are three. The inadequate state can have more than one conflict; we can run into more inadequate states while constructing the LAR automaton; and one or more simulated stacks may grow indefinitely, so the FS look-ahead automaton construction process may not terminate, generating more and more states.

The first two problems are easily solved. If the inadequate LR state has more than one conflict, we start a separate level in our initial LAR state for each possible action. Again states in which all levels but one are empty are terminal states (of the LAR automaton). And if we encounter an inadequate state p_x in the simulated stack of level l, we just copy that stack for all actions that p_x allows, keeping all copies in level l. Again states in which all levels but one are empty are terminal states; we do not need to find out which of the stacks in that level is the correct one.

The problem of the unbounded stack growth is more interesting. Consider the grammar of Figure 9.51; it produces the same language as that of Figure 9.46, but is right-recursive rather than left. The pertinent part of the LR(0) automaton is shown in Figure 9.52.

We process the first two **d**s just as above:

```
S_S  →  A | B
A    →  C A | a
C    →  d
B    →  D B | b
D    →  d
```

Fig. 9.51. An LAR(1) grammar

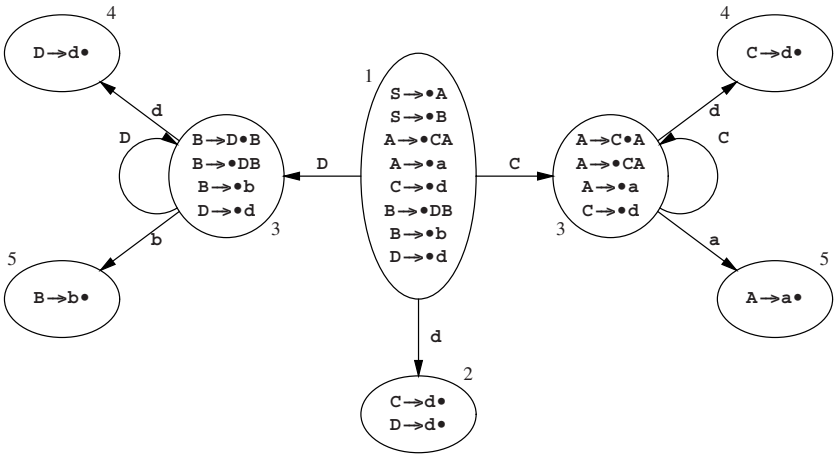
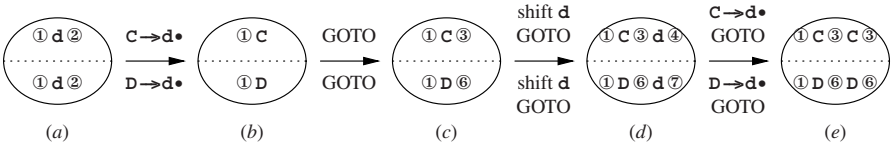
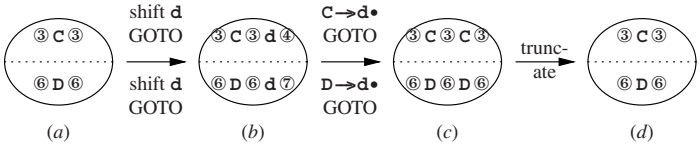


Fig. 9.52. Part of the LR(0) automaton for the grammar of Figure 9.51



but to our dismay we see that the miracle of the identical states does not repeat itself. In fact, it is easy to see that for each subsequent **d** the stacks will grow longer, creating more and more different LAR states, preventing us from constructing a *finite*-state look-ahead automaton at parser generation time. Bermudez and Schimpf's solution to this problem is simple: keep the top-most m symbols of the stack only. This leads to *LAR(m) parsing*. Note that, although we are constructing look-ahead automata, the m is not the length of the look-ahead, but rather the amount of left context maintained while doing the look-ahead. If the resulting LAR automaton has loops in it, the look-ahead itself is unbounded, unrelated to the value of m.

Using this technique with $m = 1$ truncates the stacks of frame *e* above to those in frame *a* below:



Proceeding as before, we shift in the **d**s, perform reductions and GOTOs, and finally truncate again to $m = 1$, and we are happy to see that this leads us back to the previous state. Since there are only a finite number of stacks of maximum length m, there are only a finite number of possible states in our LAR automaton, so the construction process is guaranteed to terminate. The result for our grammar is shown in Figure 9.53.

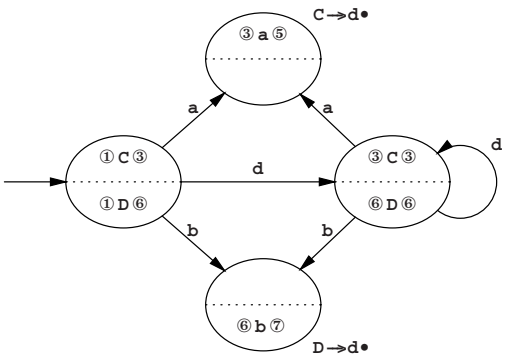


Fig. 9.53. The LAR(1) automaton for the inadequate state 2 in Figure 9.52

This technique seems a sure-fire way to resolve any problems with inadequate states, but of course it isn't. The snag is that when reducing a simulated stack we may have to reduce more symbols than are available on that stack. If that happens, the grammar is not LAR(m) — so the fact that our above attempt with $m = 1$ succeeded proved that the grammar of Figure 9.51 is LAR(1). Making m larger than the length of the longest right-hand side does not always help since successive reduces may still shorten the stack too much.

The above procedure can be summarized as follows:

- For each inadequate state p_x we construct an LAR look-ahead automaton, which starts in a provisional LAR state, which has as many levels as there are possible actions in p_x.
- In each provisional state we continue to perform reduce and GOTO actions until each stack has an LR state which allows shifting on top, all the while truncating the stack to m symbols.
  - If we run into an inadequate state p_y in this process, we duplicate the stack inside the level and continue with all actions p_y allows.

  - If we have to reduce more symbols from a stack than it contains, the grammar is not LAR(m).
  - If we shift over the end marker **#** in this process, the grammar is ambiguous and is not LAR(m).

  If there is now only one non-empty level left in the LAR state, it is a terminal LAR state. Otherwise the result is either a new LAR state, which we process, or a known LAR state.
- For each new LAR state p we create transitions $p \xrightarrow{t} p_t$ for all tokens t that p allows, where the p_t are new provisional states.
- We continue the above process until there are no more new LAR states or we find that the grammar is not LAR(m).

We regret to say that we have again left a couple of complications out of the discussion. When working with $m > 1$, the initial LAR state for an inadequate LR state must contain stacks that derive from the left context of that state. And the number of LAR automata can be reduced by taking traditional LALR look-ahead into account. These complications and more are discussed by Bermudez and Schimpf [82], who also provide advice about obtaining reasonable values for m.

9.14 LR(k) as an Ambiguity Test

It is often important to be sure that a grammar is not ambiguous, but unfortunately that property is undecidable: it can be proved that there cannot be an algorithm that can, for every CF grammar, decide whether it is ambiguous or unambiguous. This is comparable to the situation described in Section 3.4.2, where the fundamental impossibility of a recognizer for Type 0 grammars was discussed. (See Hopcroft and Ullman [391, p. 200]). The most effective ambiguity test for a CF grammar we have at present is the construction of the corresponding LR(k) automaton, but it is not a perfect test: if the construction succeeds, the grammar is guaranteed to be unambiguous; if it fails, in principle nothing is known. In practice, however, the reported conflicts will often point to genuine ambiguities.

The construction of an LR-regular parser (Section 9.13.2) is an even stronger, but more complicated test; see Heilbrunner [392] for a precise algorithm. Schmitz and Farré [398] describe a different very strong ambiguity test that can be made arbitrarily strong at arbitrary expense, but it is experimental.

9.15 Conclusion

The basis of bottom-up parsing is reducing the input, through a series of sentential forms, to the start symbol, all the while constructing the parse tree(s). The basis of deterministic bottom-up parsing is finding, with certainty, in each sentential form a segment α equal to the right-hand side of a rule $A \rightarrow \alpha$ such that the reduction using that rule will create a node A that is guaranteed to be part of the parse tree. The basis

of left-to-right deterministic bottom-up parsing is finding, preferably efficiently, the leftmost segment with that property, the *handle*.

Many plans have been devised to find the handle. Precedence parsing inserts three types of marker in the sentential form: $\lessdot$ for the left end of a handle; $\doteq$ for use in the middle of a handle; and $\gtrdot$ for the right end of the handle. The decision which marker to place in a given position depends on one or a few tokens on the left and on the right of the position. Bounded context identifies the handle by a left and right context, each a few tokens long. LR summarizes the entire left context into a single state of an FSA, which state then identifies the reduction rule, in combination with zero, one, or a few tokens of the right context. LR-regular summarizes the entire right context into a single state of a second FSA, which state in combination with the left context state then identifies the reduction rule. Many different FSAs have been proposed for this purpose.

Problems

Problem 9.1: Arguably the simplest deterministic bottom-up parser is one in which the shortest leftmost substring in the sentential form that matches a right-hand side in the grammar is the handle. Determine conditions for which this parser works. See also Problem 10.9.

Problem 9.2: Precedence parsing was explained as "inserting parenthesis generators". Sheridan [111] sketches an algorithm that inserts sufficient numbers of parentheses. Determine conditions for which this works.

Problem 9.3: There is an easy approach to LR(0) automata with shift/reduce conflicts only: shift if you can, reduce otherwise. Work out the consequences.

Problem 9.4: Extend the tables in Figure 9.18 for the case that the input consists of sentential forms containing both terminal and non-terminal symbols rather than strings of terminals. Same question for Figure 9.28.

Problem 9.5: Complete the LR(2) ACTION and GOTO tables of Figure 9.33.

Problem 9.6: Design the combined LR($k = 0, 1, > 1$) algorithm hinted at on page 299.

Problem 9.7: Devise an efficient table structure for an LR(k) parser where k is fairly large, say between 5 and 20. (Such grammars may arise in grammatical data compression, Section 17.5.1.)

Problem 9.8: An LR(1) grammar is converted to CNF, as in Section 4.2.3. Is it still LR(1)?

Problem 9.9: In an LR(1) grammar in CNF all non-terminals that are used only once in the grammar are substituted out. Is the resulting grammar still LR(1)?

Problem 9.10: Is it possible for two items in the LALR(1) channel algorithm to be connected both by propagation channels and by spontaneous channels?

Problem 9.11: Apply the algorithm of Section 9.7.1.3 to the grammar of Figure 9.30.

Problem 9.12: The **reads** and **directly-reads** relations in Section 9.7.1.3 seem to compute the FIRST sets of some tails of right-hand sides. Explore the exact relationship between **reads** and **directly-reads** and FIRST sets.

Problem 9.13: *Project for Prolog fans:* The relations in the algorithm of Section 9.7.1.3 immediately suggest Prolog. Program the algorithm in Prolog, keeping the single-formula formulation of page 310 as a single Prolog clause, if possible.

Problem 9.14: Although the LALR-by-SLR algorithm as described by Bermudez and Logothetis [79] can compute look-ahead sets of reduce items only, a very simple modification allows it to compute the LALR look-aheads of any item. Use it to compute the LALR look-ahead sets of `E→E•-T` in states 4 and 9 of Figure 9.25.

Problem 9.15: *Project:* The channels in the channel algorithm in Section 9.7.1.2 and the relations in the relations algorithm in Section 9.7.1.3 bear some resemblance. Work out this resemblance and construct a unified algorithm, if possible.

Problem 9.16: *Project:* It is not obvious that starting the FSA construction process in Section 9.10.2 from state s_0 yields the best possible set, either in size or in amount of stack activity saved. Research the possibility that a different order produces a better set of FSAs, or even that a different or better set exists that does not derive from some order.

Problem 9.17: Derive left-context regular expressions for the states in Figure 9.17 as explained in Section 9.12.

Problem 9.18: Write a program to construct the regular grammar for the left contexts of a given grammar.

Problem 9.19: Write a program to construct the regular grammar for the right contexts of a given grammar.

Problem 9.20: It seems reasonable to assume that when the dot right contexts in a given inadequate state have a non-empty intersection even on the regular expression level, the grammar must be ambiguous: there is at least one continuation that will satisfy both choices, right up to the end of the input, and thus lead to two successful parses. The grammar `S→aSa | a`, which is unambiguous, proves that this is not true: `D_2S→•a=aaa*$` and `D_2S→a•=aa*$`, and they have any string in `aaa*$` in common. What is wrong with the reasoning?

Problem 9.21: Construct a grammar that has a DR automaton with a loop in it.

Problem 9.22: Since the regular envelope in LR-regular parsing is too wide, it can happen that the rest of the input is inside the regular envelope but outside the CF right context grammar it envelopes. What happens in this case?

Problem 9.23: Show that the naive implementation of the LR-regular parser in Section 9.13.2 indeed has a time requirement of $O(n^2)$.

Problem 9.24: Work out the details of building a reverse FSA $\overleftarrow{F}$ from a given FSA F, both when F is non-deterministic and when it is already deterministic. ($\overleftarrow{F}$ should recognize the reverse of the strings F recognizes.)

Problem 9.25: Derive a deterministic automaton (or a regular expression) for `T` from the automaton in Figure 9.48.

Problem 9.26: Devise a way to do the transformation to a regular envelope on the deterministic LR(0) automaton (for example Figure 9.17) rather than on the nondeterministic one.

Problem 9.27: 1. Make the NFA in Figure 9.48 deterministic for `T`. 2. Derive a regular expression for `T` and use it in the expression `[-T|)]*)[-T]*$` derived for the item right context of `E->•E-T` in state 6 in Section 9.12.2.

Problem 9.28: *Project*: Design reasonable error reporting for an LR-regular parser. (Background: If the backward scan of an LR-regular parser is performed on incorrect input, chances are that the automaton gets stuck somewhere, say at a position *P*, which means that no look-aheads will be attached to any positions left of *P*, which in turn means that parsing cannot even start. Giving an error message about position *P* is unattractive because 1) it may not be the leftmost error, which is awkward if there is more than one error, and 2) no reasonable error message can be given since there is finite-state information only.)

Problem 9.29: *Project Formal Languages:* The argument on page 328 suggesting that it is undecidable whether a grammar is LR-regular or not works the wrong way: it reduces our problem to an undecidable problem, but it should reduce an undecidable problem to ours. Correct.

Problem 9.30: On page 337 we write that the grammar is not LAR(*m*) if during reducing a simulated stack we have to reduce more symbols than are available on that stack. But why is that a problem? We know which reduction to do, so we could just do it. Or can we?

10

Non-Canonical Parsers

Top-down parsers make their predictions in pre-order, in which the parent nodes are identified *before* any of their children, and which imitates leftmost derivations (see Section 6.1); bottom-up parsers perform their reductions in post-order, in which the parent nodes are identified *after* all of their children have been identified, and which imitates rightmost derivation (see the introduction in Chapter 7). These two orders of producing and visiting trees are called "canonical", and so are the parsing techniques that follow them.

Non-canonical parsing methods take liberties with these traditional orders, and sometimes postpone the decisions that would be required to create parse trees in pure pre- or post-order. This allows them to use a larger set of grammars, but on the other hand these methods create fragments of parse trees, which have to be combined at later moments.

Like their canonical counterparts, non-canonical methods can be classified as top-down (Section 10.1) and bottom-up (Section 10.2) methods, based on whether they primarily use pre-order or post-order. There are deterministic and general non-canonical methods. The deterministic methods allow parsing in linear-time; as with LL and LR methods, they can be generalized by applying a limited breadth-first search. Altogether the non-canonical methods form a large and diverse field that has by no means been completely explored yet.

Figure 10.1 shows the relation between non-canonical parsing and the corresponding non-canonical production process, as Figure 3.9 did for canonical parsing. In this case just the node for `q` has been identified. Again the dotted line represents the sentential form.

The most important property of deterministic non-canonical parsing is that it allows a larger class of grammars to be used without modification while retaining linear time requirements. Since it has simultaneously aspects of top-down and bottom-up parsing it can also provide further insight in parsing; see, for example, Demers [103], who describes a parsing technique on a gliding scale between LL(1) and SLR(1).

On the down side there is an increased complexity and difficulty, both for the implementer and the user. Postponing decisions does not come free, so non-canonical parsing algorithms are more complex and require more ingenuity than their canonical

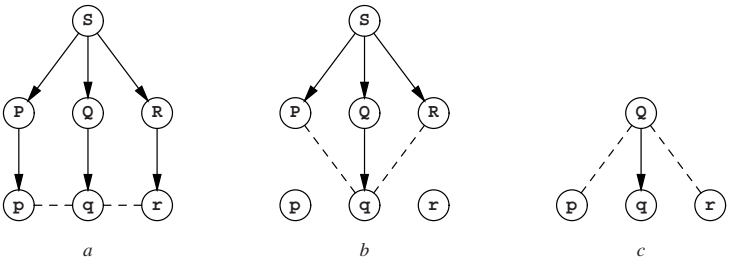


Fig. 10.1. Full parse tree (*a*), non-canonical top-down (*b*), non-canonical bottom-up (*c*)

counterparts; this makes them less attractive for implementers. And where LL and LR parser generators can leave the construction of the parse tree to the user, the non-canonical methods identify parse tree nodes in an often unintuitive order, making it next to impossible for users to construct a parse tree on their own. Consequently we find non-canonical methods primarily in systems that offer the user a finished parse tree.

Creating a node in a parse tree is actually a two-step process: first the node is created and then it is identified, i.e., labeled with a non-terminal. In almost all parsers these two steps coincide, but that does not have to be. Some non-canonical parsers create the nodes, together with their identifications, in non-canonical order; others create the nodes in canonical order, but identify them later, in some different order. Examples of the latter are PLL(1) and Partitioned LR. There should probably be different names for these approaches but there are not, to our knowledge. Note that operator-precedence creates the nodes in canonical order but identifies them later, or not at all, and therefore borders on the non-canonical.

10.1 Top-Down Non-Canonical Parsing

Top-down non-canonical parsers postpone parse tree construction decisions, but not as far as canonical bottom-up parsers. As a result they are less powerful but often allow earlier decisions than LR parsers. This is important when early semantics actions are desired.

We will discuss here three deterministic top-down non-canonical methods: left-corner parsing, cancellation parsing, and Partitioned LL. The first two allow top-down parsing with left-recursive grammars, while the third allows grammars for languages that would require unbounded look-ahead in traditional top-down parsing.

10.1.1 Left-Corner Parsing

As we have seen in Section 6.3.2, a standard top-down parser cannot deal with left-recursive grammars, since it has no way of knowing how many rounds of left recursion it should predict. Suppose we postpone that decision and concentrate on predicting a suitable subtree; once we have found that, we may be able to decide

whether another round of left recursion is needed. This leads to a technique called "left-corner parsing".

10.1.1.1 Left Spines

Consider the grammar for simple arithmetic expressions in Figure 10.2 copied from Figure 9.2, and the input string **n+n×n#**, where **#** is the usual end marker.

```
S_S  →  E
 E   →  E + T
 E   →  T
 T   →  T × F
 T   →  F
 F   →  n
 F   →  ( E )
```

Fig. 10.2. A grammar for simple arithmetic expressions

We start with the prediction **S#** and the first token of the input, **n**. A traditional LL parser would want to produce the complete left spine of the parse tree before the **n** is matched. It would use something like the following reasoning: the pair (**S**,**n**) predicts an **E**; the pair (**E**,**n**) either predicts **E+T**, which brings us back to **E**, or a **T**; the pair (**T**,**n**) predicts **T×F**, which brings us back to **T** or an **F**; and the **F** finally predicts the **n**. Only then can matching take place. But the LL parser cannot do so deterministically, since it cannot decide how many **E+T**s and **T×F**s it should predict, as shown in Figure 10.3(*a*).

A *left-corner parser* postpones these decisions, finds that the left-corner prediction **F→n** is a decision that can be made with certainty, and is satisfied with that. The predicted **F** is then parsed recursively (see below). By the same reasoning, the resulting **F** can only derive from a prediction **T→F**, which allows the **F** to be matched. This allows us to look behind the **T**, where we see a **+**. This **+** tells us that the parse tree node starting with the **T** cannot have been **T×F**, but must have derived from **E→T**. So we predict **E→T**, match the **T** and look behind the **E**, where the **+** tells us to predict **E→E+T**. The predicted **+** and the **+** from the input are now matched, and the left-corner parser starts parsing **n×n** with **T** as the prediction.

Several points must be made here. The first is that in some weird way we have been making predictions from hindsight; we will return to this point further on. The second is that the above process fixed the left spine of the parse tree to some extent, but not completely; see Figure 10.3(*b*). The reason is of course that the parser cannot know yet if more nodes **E+T** will be required; they would be if the input were, for example, **n+n×n+n+n**. The third is that the **F** (and the **T** and the **E**) must be parsed recursively, since they may match large portions of the input string. If the input had been **(n+n×n+n)+n**, the whole sub-expression between parentheses would have to be absorbed by the parsing of the **F**, to allow us to look behind it. And the fourth, and

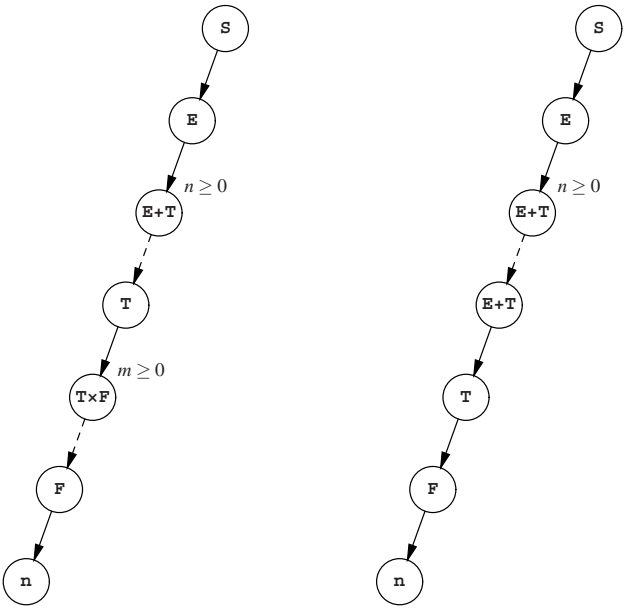


Fig. 10.3. Left spines: initially (*a*) and after some nodes have been recognized (*b*)

probably most important point is that the above sketch is not yet a usable algorithm and needs more detail.

10.1.1.2 The Production Chain Automaton

Figure 10.3(*a*) provides more information than we have given it credit for: it shows clearly that the left spine is described by a regular expression: **`SE(E+T)`*****`T(T×F)`*****`Fn`**. This regular expression corresponds to a finite-state automaton, which is depicted in Figure 10.4(*a*), and which also shows the second alternative for **`F`**, **`(E)`**. The nodes are the predicted non-terminals that participate in the left-corner process, and the arrows are labeled with the rules involved in the predictions. The automaton describes all leftmost production chains from **`S`**; such an automaton is called a *production chain automaton* for **`S`** (Lomet [102]).

In the sketch of the algorithm in the previous section we were interested in the token that became visible behind the first non-terminal in a prediction; they form the look-ahead sets of the predictions and are also shown in the picture of the automaton. The non-terminal **`S`** starts of with a look-ahead set of [**`#`**], and the production **`S→E`** passes it on to the **`E`**. So along the **`S`**-to-**`E`** arc the look-ahead set is [**`#`**]. The production rule **`E→E+T`** adds a **`+`** to this set, so when the production rule **`E→T`** is taken, the look-ahead set has grown to [**`#+`**]. In the same way **`T→T×F`** adds a ×, so when the production rule **`T→F`** is taken, the look-ahead set is [**`#+×`**]. The final predictions

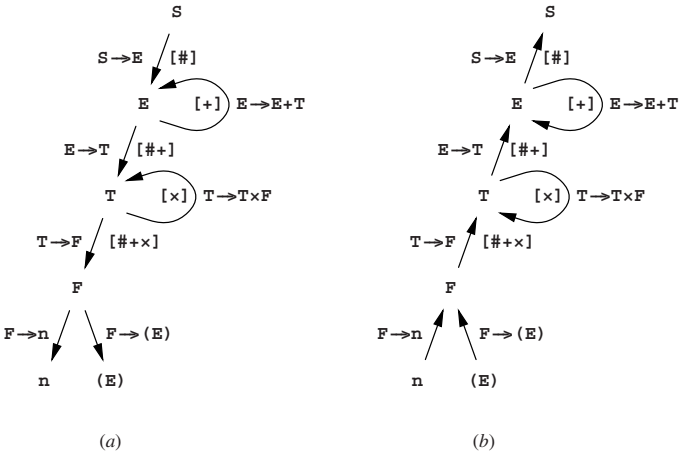


Fig. 10.4. Left spine finite-state automata, predictive (*a*) and reversed (*b*)

F→n and **F→(E)** do not have look-ahead sets, since these rules start with the terminal symbols and are identified by these.

The automaton as depicted in Figure 10.4(*a*) generates the predictions from **S** to the first terminal symbol non-deterministically, since there is nothing to guide the automaton into determinacy. But in our sketch of the algorithm we used the look-ahead tokens discovered after the first non-terminal of a rule to find our way *backwards* through the automaton. This means that if we reverse the arrows in the production automaton, we can let ourselves be guided by the look-ahead tokens to find the next (actually the previous!) prediction. When the reversed automaton of each non-terminal in a grammar *G* is deterministic, we have a deterministic parser, and *G* is of type *LC(1)*.

We see in Figure 10.4(*b*) that the reverse production chain automaton for our grammar is indeed deterministic. The working of the parser on **n+n×n** is now more easily followed. Starting at the bottom, the initial **n** of the input gives us the prediction **F→n** and absorbing the **F** reveals the **+** after it. This **+** leads us to the prediction **T→F**, where the path splits in a path labeled **[#+]** and one labeled **[×]**. Since the input symbol is a **+**, we follow the first path and predict a parse tree node **E→T**. At the automaton node **E** the path splits again, into one labeled **[#]** and one labeled **[+]**. Since the input symbol is still the **+**, we take the second path and predict a parse tree node **E→E+T**. Now the **+** can be matched, and the parser recursively recognizes the **n×n** with prediction **T**, as described below. When that is done, we are back at the **E** node in the FSA, but now the input symbol is **#**. So we predict **S→E**, create a parse tree node **S**, match the **E**, and the parsing is finished.

If the input had been the erroneous string **n)**, the automaton would have got stuck at the **F** since the input symbol **)** is not in the look-ahead set of the prediction **T→F**, and the parser would have reported an error.

To parse the remaining **n×n** recursively with prediction **T**, a production chain automaton for **T** is created, using the same technique as above. Since it starts with a look-ahead set of **[#+]** just as the **T** in Figure 10.4(*b*), this automaton is identical to the lower part of that figure, so no new automaton is needed. But that is not always the case, as the following example shows.

Suppose the input is **(n)**. Then we enter the automaton for **S** at the **(** at the bottom, and predict **F→(E)**. The open parenthesis is matched, and we now want to parse **n)#** with the prediction **E**. But this **E** is followed by a **)**, unlike the one in Figure 10.4(*b*), which is followed by a **#**. There are two ways to deal with this.

The first is to indeed create a new automaton for **E[)]**; this automaton is similar to the automaton in Figure 10.4(*b*) from the **E** node down, except that the token **#** in the look-ahead sets is replaced by **)**. More in general, we create a new automaton for each combination of non-terminal and look-ahead set. This approach produces a *full-LC(1)* parser, comparable to the full-LL(1) parser explained in Section 8.2.3.

The second is to add the closing parenthesis to the look-ahead set of the **E** node in Figure 10.4(*b*) and update the automaton. If we do this for all non-terminals in the grammar, there will be only one node in the production chain automaton for each non-terminal and the look-ahead sets become equal to the FOLLOW sets. This course of action leads to a *strong-LC(1)* parser, comparable to a strong-LL(1) parser. Like the latter, it has smaller tables and weaker error detection properties than its full-LC(1) counterpart. For example, the erroneous input **n)** will fool it into predicting **F→n**, **T→F** and **E→T**, and only then will the automaton get stuck on an input symbol **)** and a look-ahead set **[#]**.

10.1.1.3 Combining the Chain Automaton and the Parse Stack

The actions of a left-corner parser can be implemented conveniently as a top-down parser, using a stack and a parse table. The table is indexed by the first token of the prediction if it is a non-terminal, and the first token of the rest of the input, the look-ahead. We start with the prediction **S#**; see Figure 10.5. We have seen above that the non-terminal **S** and the look-ahead **n** lead to the prediction $\mathbf{F}_1 \rightarrow \mathbf{n}$, so we stack the **n** and add the $\mathbf{F}_1$ to the analysis (we have appended the subscript $_1$ to identify the first rule for **F**). But that cannot be all: this way we lose the part of the prediction between the **F** and the **S** in the automaton of Figure 10.4(*b*). So we also stack a new-made symbol **S\F**, to serve as a source for predictions for the left spine from **S** to **F**. As a stack symbol, **S\F** matches the rest of **S** after we have matched an **F**; in other words, any string produced by **F S\F** is a terminal production of **S**.

Next we match the **n**, adding it to the analysis. Now we need a prediction for **S\F**, with look-ahead **+**, and the automaton tells us to predict $\mathbf{T}_2 \rightarrow \mathbf{F}$. We add the $\mathbf{T}_2$ to the analysis, but we have parsed the **F** already, so we do not stack it; it is in fact the left operand of the $\mathbf{T}_2$ in the analysis. We do, however, stack a symbol **S\T**, to cover the rest of the left spine. The symbol **S\T** designates the position **T** in the automaton for **S**, the look-ahead is still **+**, so the automaton wants us to predict $\mathbf{E}_2 \rightarrow \mathbf{T}$. Like before, we have already parsed the entire right-hand side of the prediction, so we only stack the symbol **S\E**, which brings us to the fifth frame in Figure 10.5. Now

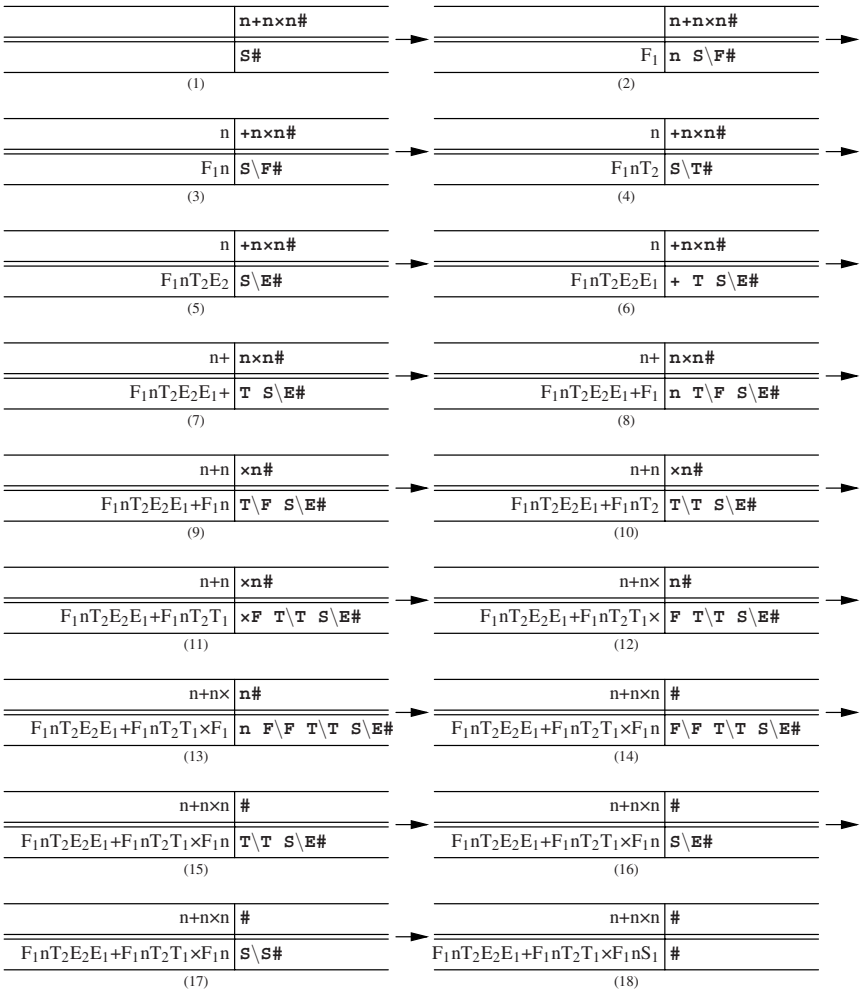

n+n×n# | S#
(1)
n+n×n# | F1 n S\F#
(2)
n | +n×n# | F1n | S\F#
(3)
n | +n×n# | F1nT2 | S\T#
(4)
n | +n×n# | F1nT2E2 | S\E#
(5)
n | +n×n# | F1nT2E2E1 | + T S\E#
(6)
n+ | n×n# | F1nT2E2E1+ | T S\E#
(7)
n+ | n×n# | F1nT2E2E1+F1 | n T\F S\E#
(8)
n+n | ×n# | F1nT2E2E1+F1n | T\F S\E#
(9)
n+n | ×n# | F1nT2E2E1+F1nT2 | T\T S\E#
(10)
n+n | ×n# | F1nT2E2E1+F1nT2T1 | ×F T\T S\E#
(11)
n+n× | n# | F1nT2E2E1+F1nT2T1× | F T\T S\E#
(12)
n+n× | n# | F1nT2E2E1+F1nT2T1×F1 | n F\F T\T S\E#
(13)
n+n×n | # | F1nT2E2E1+F1nT2T1×F1n | F\F T\T S\E#
(14)
n+n×n | # | F1nT2E2E1+F1nT2T1×F1n | T\T S\E#
(15)
n+n×n | # | F1nT2E2E1+F1nT2T1×F1n | S\E#
(16)
n+n×n | # | F1nT2E2E1+F1nT2T1×F1n | S\S#
(17)
n+n×n | # | F1nT2E2E1+F1nT2T1×F1nS1 | #
(18)

Fig. 10.5. Left-corner parsing of the input **n+n×n**

the automaton tells us to predict **E**$_1$ →**E+T**, of which we have already recognized the left corner **E**. So we stack the **+T**, and of course our reminder, again **S****E**, after which the **+** is matched.

Now we have again a "normal" non-terminal on top of the prediction stack: **T**. In principle we would now need a new automaton for **T**, but since **T** occurs in the automaton for **S**, that occurrence will serve. The recognition of the **n** as a **F**$_1$ and the **F** as a **T**$_2$ mimic the sequence of events above, but now something happens that requires our attention: after having recognized a **T** while looking for a **T**, we cannot just pack up and be satisfied with it, but we have to acknowledge the possibility that we have to stay in the automaton for **T** for another round, that there may be more of

T after this **T**. We do this by stacking the symbol for "the rest of **T** after **T**", **T****T**. This symbol allows two predictions: leave the automaton, and stay in it.

We see immediately how necessary it was to stack the **T****T** symbol, since we *do* have to stay in the automaton for **T**, to parse the subsequent **×n** (frames 10 through 15). Only in frame 15 can we decide that we have seen the whole **T** we were looking for, due to the look-ahead **#**, and leave the **T** automaton.

Something similar happens in frame 17, where we stop at the top of the automaton for **S** on the symbol **S****S**, considering whether to continue looking for more **S** or to stop here. But since there is no way to stay in the automaton in the state **S****S**, the transition to frame 18 is automatic, and the parsing is finished.

We can collect our decisions in a parse table. The complete table is given in Figure 10.6; since we did not construct separate automata for **E**, **T**, and **F** for different look-aheads, it represents a strong-LC(1) parser for the grammar of Figure 10.2.

	n	×	+	(	)	#
S	**n S****F**			**(E) S****F**		
S**F**		**S****T**	**S****T**			**S****T**
S**T**		**× F S****T**	**S****E**			**S****E**
S**E**			**+ T S****E**			**S****S**
S**S**						ε
E	**n E****F**			**(E) E****F**		
E**F**		**E****T**	**E****T**		**E****T**	**E****T**
E**T**		**× F E****T**	**E****E**		**E****E**	**E****E**
E**E**			**+ T E****E**		ε	ε
T	**n T****F**			**(E) T****F**		
T**F**		**T****T**	**T****T**			**T****T**
T**T**		**× F T****T**	ε		ε	ε
F	**n F****F**			**(E) F****F**		
F**F**		ε	ε		ε	ε

Fig. 10.6. LC parse table for the grammar of Figure 10.2

10.1.1.4 Obtaining A Parse Tree

The last frame of Figure 10.5 shows the left-corner analysis of **n+n×n** to be $\mathbf{F}_1\mathbf{n}\mathbf{T}_2\mathbf{E}_2\mathbf{E}_1\mathbf{+}\mathbf{F}_1\mathbf{n}\mathbf{T}_2\mathbf{T}_1\mathbf{\times}\mathbf{F}_1\mathbf{n}\mathbf{S}_1$ but that is less helpful than one would hope. The reason is of course that non-canonical methods identify the nodes in the parse tree in an unusual order, in this case in infix order. As explained in Section 3.1.3, infix order requires parentheses to be unambiguous, but the presented analysis does not include them (see Problem 10.3). So the analysis has to be analysed further to yield a parse tree.

To properly parenthesize the analysis, we need to identify the left and right children of each node.

- A grammar rule whose right-hand side starts with a terminal or is empty is recognized before all of its children. For example, a node for the second rule for **F**, **F→(E)**, has zero left children and 3 right children; we will represent the node as $\mathbf{F}_2$**[0+3]**.
- A grammar rule whose right-hand side starts with a non-terminal is recognized after its first child and before the rest of its children. For example, a node for the first rule for **T**, **T→T×F**, has one left child and 2 right children; we will represent the node as $\mathbf{T}_1$**[1+2]**.

Using these rules we can parenthesize the left-corner analysis of **n+n×n** as shown in Figure 10.7.

```
(
    (((F1[0+1] n) T2[1+0]) E2[1+0])
    E1[1+2]
    +
    (
        ((F1[0+1] n) T2[1+0])
        T1[1+2]
        ×
        (F1[0+1] n)
    )
S1[1+0]
)
```

Fig. 10.7. The fully parenthesized infix left-corner analysis of **n+n×n**

Now having to "analyse the analysis" may raise eyebrows, but actually the problem is caused by the linear format of the analysis shown in diagrams like Figure 10.5. Left-corner parser generators of course never linearize the parse tree in this way but rather construct it on the fly, using built-in code that gathers the children of nodes as itemized above, so the problem does not materialize in practice.

10.1.1.5 From LC(1) to LL(1)

The recognizing part of the left-corner parser presented above is indistinguishable from that of an LL(1) parser. Its stack, parse table, and mode of operation are identical to those of an LL(1) parser; only the construction of the parse table from the grammar and the construction of the analysis differ. This suggests that there is an LL(1) grammar that gives the same language as the LC(1) grammar, and in fact there is.

An LL(1) grammar for the language produced by an LC(1) grammar can always be obtained by the following technique, for which we need the notion of "left-spine child". A non-terminal B is a *left-spine child* of A (written $B \angle A$) if there is a grammar rule $A \rightarrow B\alpha$ for some possibly empty α or if there is a rule $A \rightarrow C\beta$ and $B \angle C$. So B is a left-spine child of A if B can occur on a left spine starting at A. We shall also need

to define that A is a left-spine child of itself: $A \angle A$. More in particular, in our example we have **S**$\angle$**S**$\angle$**E**$\angle$**E**$\angle$**T**$\angle$**T**$\angle$**F**$\angle$**F**. The $\angle$ is also called the *left-corner relation*.

To construct the LL(1) grammar, we start from the root non-terminal of the LC(1) grammar, **S** in our example, and we are going to need one or more rules for it in the LL(1) grammar. To find rules for a non-terminal A in the LL(1) grammar, we look for rules in the LC(1) grammar with a left-hand side B that is a left-spine child of A and whose right-hand side starts with a terminal or is empty. Such rules are of the form $B \rightarrow \beta$, where β starts with a terminal or is ε. This β can be a left-corner for A, after which we still have to parse the part of A after B, that is, $A \backslash B$. So for each such $B \rightarrow \beta$ we add a rule $A \rightarrow \beta\, A \backslash B$ to the LL(1) grammar. For our start symbol **S** there are two such rules, **F**$\rightarrow$**n** and **F**$\rightarrow$**(E)**, which give us two rules for the LL(1) grammar: **S**$\rightarrow$**n S**$\backslash$**F** and **S**$\rightarrow$**(E) S**$\backslash$**F**.

So now we need one or more rules for **S**$\backslash$**F** in the LL(1) grammar. We obtain these in a similar but slightly different way. To find rules for a non-terminal $A \backslash B$ in the LL(1) grammar, we look for rules in the LC(1) grammar with a left-hand side C that is a left-spine child of A and whose right-hand side starts with the non-terminal B. Such rules are of the form $C \rightarrow B\gamma$, where γ may be empty. Since we have already parsed the B prefix of A, and C is a left-corner child of A, we can try to continue by parsing γ, and if we succeed, we will have parsed a prefix of A produced by C, which leaves $A \backslash C$ to be parsed. So for each such $C \rightarrow B\gamma$ we add a rule $A \backslash B \rightarrow \gamma\, A \backslash C$ to the LL(1) grammar. For **S**$\backslash$**F** there is only one such rule, **T**$\rightarrow$**F**, which causes us to add **S**$\backslash$**F**$\rightarrow$**S**$\backslash$**T** to the LL(1) grammar.

For **S**$\backslash$**T**, however, there are two such rules in the LC(1) grammar, **T**$\rightarrow$**T**$\times$**F** and **E**$\rightarrow$**T**, resulting in two rules in the LL(1) grammar, **S**$\backslash$**T**$\rightarrow\times$ **F S**$\backslash$**T** and **S**$\backslash$**T**$\rightarrow$**S**$\backslash$**E**. A similar step lets us create two rules for **S**$\backslash$**E**: **S**$\backslash$**E**$\rightarrow+$ **F S**$\backslash$**E** and **S**$\backslash$**E**$\rightarrow$**S**$\backslash$**S**. The latter, deriving from $\mathbf{S}_s \rightarrow$**E**, requires a rule for **S**$\backslash$**S**. Rules of the form $A \backslash A \rightarrow \varepsilon$ can always be created for any A when required, so we add a rule **S**$\backslash$**S**$\rightarrow\varepsilon$ to the LL(1) grammar.

The above new rules have introduced the non-terminals **E**, **T**, and **F** into the LL(1) grammar, and rules for these must also be created, using the same patterns. Figure 10.8 shows the final result. We see that in addition to the trivial rule **E**$\backslash$**E**$\rightarrow\varepsilon$ there is another rule for **E**$\backslash$**E**, reflecting the fact that **E** is directly left-recursive. The same applies to **T**$\backslash$**T**.

The above transformation can be performed on any CF grammar. If the result is an LL(1) grammar the original grammar was LC(1). If we offer the grammar to a strong-LL(1) parser generator, we obtain a strong-LC(1) parser; using a full-LL(1) parser generator yields a full-LC(1) parser.

Deterministic left-corner parsing was first described extensively in 1970 by Rosenkrantz and Lewis, II [101]. A non-deterministic version was already used implicitly by Irons in 1961 [2]. It seems a good candidate for a model of human natural language parsing (see, for example Chester [377], Abney and Johnson [383], and Resnik [384]).

```
  S   →  n S\F | ( E ) S\F
S\F   →  S\T
S\T   →  S\E | × F S\T
S\E   →  + T S\E | S\S
S\S   →  ε

  E   →  n E\F | ( E ) E\F
E\F   →  E\T
E\T   →  E\E | × F E\T
E\E   →  + T E\E | ε

  T   →  n T\F | ( E ) T\F
T\F   →  T\T
T\T   →  × F T\T | ε

  F   →  n F\F | ( E ) F\F
F\F   →  ε
```

Fig. 10.8. LL(1) grammar corresponding to the LC(1) grammar of Figure 10.2

10.1.2 Deterministic Cancellation Parsing

The non-deterministic version of cancellation parsing described in Section 6.8 can, as usual, be made deterministic by equipping it with look-ahead information.

As with left-corner parsing, the look-ahead set of a non-terminal *A* consists of two components: the FIRST sets of the non-left-recursive alternatives of *A*, and the set of tokens that can follow *A*. As with LL(1) and LC(1) techniques, there are two possibilities to compute the follow part of the look-ahead set: it can be computed separately for each occurrence of *A*, or it can be replaced by the FOLLOW set of *A*. But in cancellation parsing there is another influence on the look-ahead sets: the set of alternatives of *A*, since that set is not constant but depends on *A*'s cancellation set; so the FIRST sets also fluctuate. There are again two possibilities here. We can compute these "first" look-ahead sets separately for each combination of *A* and its possible cancellation sets, or we can compute it for *A* in general, disregarding the influence of the cancellation set.

So in total there are four combinations. Nederhof [105] calls the deterministic parsers resulting from separate computation of the look-ahead sets for each occurrence of each combination of a non-terminal with one of its possible cancellation sets *C(k)* parsers; using FOLLOW sets but keeping the differences in cancellation sets gives *strong-C(k)* parsers; also disregarding the cancellation sets gives the *severe-C(k)* parsers. The fourth combination, ignoring the cancellation sets but distinguishing different occurrences of the same non-terminal, is not described in the paper because it is identical to full-LL.

Since the construction of deterministic cancellation parsers is quite complicated; since such parsers are less powerful than left-corner parsers; and since non-

deterministic cancellation parsers are much more useful than their deterministic versions, we will not discuss their construction here further.

10.1.3 Partitioned LL

LL(1) parsing requires us to choose between the alternatives of a non-terminal right at the start. *Partitioned LL(1)* (or *PLL(1)*) tries to postpone this decision as long as possible, but requires that the decision will be taken before or at the end of the alternatives. This assures that the deterministic nature of the parser is preserved. Partitioned LL(k) parsing was designed by Friede [196, 195].

10.1.3.1 Postponing the Predictions by Using Partitions

We will demonstrate the technique using the grammar from Figure 10.9, which pro-

```
S_S  ->  A | B
 A   ->  aAb | ab
 B   ->  aBc | ac
```

Fig. 10.9. A difficult grammar for top-down parsing

duces the language $\mathtt{a}^n\mathtt{b}^n \cup \mathtt{a}^n\mathtt{c}^n$. Sample strings are **ab**, **aacc**, and **aaabbb**; note that "mixed" strings like **aaabbc** are forbidden.

Trying to get a top-down parser for this grammar is quite interesting since the language it generates is the standard example of a language for which there cannot be a deterministic top-down parser. The reason for the inherent impossibility of such a parser for this language is that a top-down parser requires us to make a prediction for a **b** or a **c** in the second half of the sentence for each **a** we read. But the point where we can decide which of them to predict can be arbitrarily far away; and just predicting **[bc]** will not do, since that would allow "mixed" sentences like the one above.

To postpone the decision between two alternatives of a non-terminal A, we first look for a common prefix. Suppose the first alternative is $A \rightarrow \alpha\beta$ and the second is $A \rightarrow \alpha\gamma$. We can then parse the common prefix α first although we will have to predict both β and γ; we will see below how we can implement this. The moment we reach the point where the two alternatives start to differ, we try an LL(1)-like test to find out if we can see which one applies. To do this, we compute the sets FIRST(β FOLLOW(A)) and FIRST(γ FOLLOW(A)). If these are disjoint, we can base our decision on them, as we did in LL(1) parsing; we can then also discard one of the predictions.

If this were all, Partitioned LL(1) would just be LL(1) with automatic left-factoring (Section 8.2.5.2). But Partitioned LL(1) goes one step further. Suppose the LL(1) test fails and it so happens that β and γ both start with a non-terminal, say P and Q; so $\beta = P\beta'$ and $\gamma = Q\gamma'$. Partitioned LL(1) then puts P and Q together in a

"partition", which is actually a set: $\{P,Q\}$, and tries to postpone and still achieve the decision by parsing with the partition $\{P,Q\}$ as prediction. An essential requirement for this to work is of course that parsing with a partition tells in the end which member of the partition was found. We can then proceed with the proper choice of β' and γ', just as we could with β or γ after a successful LL(1) test.

At first sight this does not seem like a very bright plan, since rather than having to distinguish between the alternatives of P and Q separately, we now have to distinguish between the union of them, which will certainly not be easier. Also, when P or Q happen to be A, we get the same problem back that we had with A in the first place. But that is not true! If we now try to distinguish between the alternatives of A and run into P in one alternative and Q in the other, we can simply continue with the partition $\{P,Q\}$ — provided the rest of the problems with the parser for $\{P,Q\}$ can be solved.

10.1.3.2 A Top-Down Implementation

When we try to distinguish between the two alternatives of **S** in Figure 10.9, we find that they have no common prefix. Next we try the LL(1) test on **A** and **B**, but since both start with an **a**, it fails. So we combine **A** and **B** into **A_or_B**. The recognizing routine for **A_or_B** is shown in Figure 10.10. It requires us to handle the following set of alternatives simultaneously:

```
a A b : A
a b : A
a B c : B
a c : B
```

where the result of the recognition is given after the colon.

We see that they have a common prefix **a**, for which we construct recognizing code (**token('a');**). The alternatives are now reduced to

```
A b : A
b : A
B c : B
c : B
```

But **A** and **B** have been replaced by **A_or_B** followed by tests whether a **A** or **B** resulted. This gives the following alternatives to deal with:

```
A_or_B ( A? b : A | B? c : B)
b : A
c : B
```

where the test for the result is indicated by a question mark. Now the LL(1) test succeeds: FIRST(**A_or_B**) is **a**, which sets it off from the two other alternatives, which start with **b** and **c**, respectively. All this results in the parser of Figures 10.10 and 10.11, where we have added print statements to produce the parse tree. A sample run with input **aaaccc** yields the output

```
char A_or_B(void) {
    /* common prefix */
    token('a');
    /* decision point */
    switch (dot) {
    case 'b': token('b'); print("A->ab"); return 'A';
    case 'c': token('c'); print("B->ac"); return 'B';
    case 'a':
        switch (A_or_B()) {
        case 'A': token('b'); print("A->aAb"); return 'A';
        case 'B': token('c'); print("B->aBc"); return 'B';
        }
    default: return error("abc");
    }
}
```

Fig. 10.10. C code for the routine **`A_or_B`**

```
char S(void) {
    /* common prefix */
    /* decision point */
    switch (dot) {
    case 'a':
        switch (A_or_B()) {
        case 'A': print("S->A"); return 'S';
        case 'B': print("S->B"); return 'S';
        }
    default: return error("a");
    }
}
```

Fig. 10.11. C code for the routine **`S`**

B→ac
B→aBc
B→aBc
S→B

The recursive descent routines in a canonical LL parser just return **`true`** or **`false`**, indicating whether or not a terminal production of the predicted non-terminal was found. We see that this set of return values is extended with the identity of the non-terminal in the PLL code, and it is this small extension that makes the parser more powerful.

10.1.3.3 PLL(0) or SPLL(1)?

The original definition of PLL(k) (Friede [196]) splits the PLL(k) test into two parts: two right-hand sides $\alpha\beta$ and $\alpha\gamma$ of rules for A are distinguishable if after skipping the common prefix α at least one of the following conditions holds.

- Both β and γ start with a terminal symbol and those symbols are different.
- $\text{FIRST}_k(\beta\text{FOLLOW}_k(A))$ and $\text{FIRST}_k(\gamma\text{FOLLOW}_k(A))$ have nothing in common, where FIRST_k and FOLLOW_k are the FIRST and FOLLOW sets of length k.

This split allows the definition of PLL(0), PLL(k) with $k = 0$: the second test will always fail, but the first one remains meaningful, and saves the technique. In fact, the SLL(1) grammar from Figure 8.4 is PLL(0) under this definition — but if we replace the rule for `B` by `B→ab | aaBb` it is no longer SLL(1) while still being PLL(0), because PLL skips the common prefix. PLL(0) grammars have some theoretical significance since they are exactly the strict deterministic grammars (again Friede [196]), but their theory is simpler.

Still, $k = 0$ suggests that no look-ahead is involved, as in LR(0), where the decision about a rule can be taken on the last symbol of its production. But that is not the case here: to decide between `B→b` and `B→aBb` we need to look at the first symbol of the input. Now we could also modify the definition of LL(k), by splitting the LL(k) test as above. Then non-trivial LL(0) grammars would exist, and they would be the SLL(1) or s-grammars. So it would perhaps be more reasonable to call the PLL(0) grammars *SPLL(1) grammars* or *partitioned s-grammars*. (For LL(0) grammars under the normal LL(k) definition, see Problem 8.1.)

10.1.4 Discussion

The main advantages of canonical top-down parsing are the fact that semantic actions can be performed early in the parsing process, and the simplicity of the parser. Non-canonical top-down parsers work for more grammars, retain much of the first advantage but lose on the second.

10.2 Bottom-Up Non-Canonical Parsing

Non-canonical parsers derive their increased power from postponing some of the decisions that canonical parsers have to take. For bottom-up parsers, this immediately leads to two questions.

The first is that bottom-up parsers already postpone the recognition of a subtree (handle) to the last possible moment, after all the terminals of the handle have been read, possibly plus a number of look-ahead tokens. So what more is there to postpone? The answer is that non-canonical bottom-up methods abandon rather than postpone the hunt for the (leftmost) handle, and start looking for a subtree further on in the input. Whether this subtree can again be called a handle is a matter of definition. We will still call it a handle, although many authors reserve that term for the leftmost fully recognized subtree, and use words like “phrase” for the non-leftmost ones.

The second question is: Why is it profitable to reduce a non-leftmost handle? After all, when a non-leftmost handle has been found and the corresponding reduction

performed, the leftmost handle will still have to be found or we will never get a parse tree. Here the answer is that reducing segments further on in the input improves the look-ahead. A single token in the look-ahead may not give sufficient information to decide whether and how to reduce, but knowing that it is part of a non-terminal *A* or *B* might, and if that is the case, the grammar is amenable to non-canonical bottom-up parsing. This shows that look-aheads are essential to non-canonical bottom-up parsing, and that we will need to allow non-terminals in the look-ahead.

The non-canonical bottom-up parsing methods differ in the way they resume their search for a handle. We will show here three methods, total precedence, NSLR(1), and LR(k,∞); the bibliography in (Web)Section 18.2.2 shows several examples of other techniques.

Farré and Fortes Gálvez [209] describe a non-canonical DR(k) parser; unlike the other parsers in this chapter it can require $O(n^2)$ time to parse its input.

For the — non-canonical — BC and BPC methods see Section 9.3.1.

10.2.1 Total Precedence

Knuth [52] was the first to hint at the possibility of non-canonical bottom-up parsing, but the first practical proposal came from Colmerauer [191], who modified precedence parsing to recognize non-leftmost handles.

We shall use Colmerauer's grammar G_2 for the explanation:

$$\begin{array}{lcl} \mathtt{S}_\mathtt{S} & \rightarrow & \mathtt{a} \\ \mathtt{S} & \rightarrow & \mathtt{aSB} \\ \mathtt{S} & \rightarrow & \mathtt{bSB} \\ \mathtt{B} & \rightarrow & \mathtt{b} \end{array}$$

This grammar produces the language $\mathtt{[ab]}^n\mathtt{ab}^n$, a number of **a**s or **b**s, next an **a**, and then an equal number of **b**s. An example is **ababb**. It is tricky to make a left-to-right parser for this language, since all **a**s before the last **a** come from the rule **S→aSB**, but the last **a** comes from **S→a**, and we cannot know what is the last **a** until we have seen the end of the input, after n **b**s.

The grammar is not simple precedence (it is not LR(1) either, as is easily shown by considering the LR(1) state after the input string **aa**). Its simple-precedence table can be computed using the procedure sketched in Section 9.2.4; the result is

	#	S	B	a	b
#		⋖		⋖	⋖
S	⋗		≐		⋖
B	⋗				⋗
a	⋗	≐		⋖	⋖/⋗
b	⋗	≐		⋖	⋖/⋗

This table has two ⋖/⋗ conflicts, **a**⋖/⋗**b** and **b**⋖/⋗**b**. The first should be resolved as **a**⋗**b** when the **a** is the last **a** but as **a**⋖**b** when it is not; and the second as **b**⋖**b** when it occurs before the last **a** and as **b**⋗**b** after the last **a**. These are fundamental

conflicts, which cannot be resolved by traditional conflict resolvers (Section 9.9) or by resorting to other precedence methods.

In a *total precedence* parser we want to read on past the conflict and try to find another handle further on that will shed light on the present problem. The $\gtrdot$ relations in the conflicting entries prevents us from doing so, so we remove these; it is clear that that will have repercussion elsewhere, but for the moment it allows us to continue. Since now all combinations of **a** and **b** have the relation $\lessdot$, we will shift all **a**s and **b**s until we reach the end of the string, where we are stopped by the $\gtrdot$ in **b**$\gtrdot$**#**:

$$\texttt{\#} \lessdot \texttt{a} \lessdot \texttt{b} \lessdot \texttt{a} \lessdot \texttt{b} \lessdot \texttt{b} \gtrdot \texttt{\#}$$

This leads us to reduce the handle $\lessdot$**b**$\gtrdot$ to **B**, and this reduction turns out to be a great help.

Now that the subtree **B** has been recognized, it can be considered a newly defined terminal symbol, just as we did in cancellation parsing in Section 6.8. Going back to the simple-precedence table construction procedure sketched in Section 9.2.4, we see that the juxtaposition of **S** and **B** in the right-hand sides of **S**$\rightarrow$**aSB** and **S**$\rightarrow$**bSB** requires a $\gtrdot$ relation between all symbols in LAST_{ALL}(**S**) and **B**, if **B** is a terminal. Since LAST_{ALL}(**S**)={**a**,**b**,**B**}, we get **a**$\gtrdot$**B**, **b**$\gtrdot$**B**, and **B**$\gtrdot$**B**, which gives us the following total precedence table:

	#	S	B	a	b
#		$\lessdot$		$\lessdot$	$\lessdot$
S	$\gtrdot$		$\doteq$		$\lessdot$
B	$\gtrdot$		$\gtrdot$		$\gtrdot$
a	$\gtrdot$	$\doteq$	$\gtrdot$	$\lessdot$	$\lessdot$
b	$\gtrdot$	$\doteq$	$\gtrdot$	$\lessdot$	$\lessdot$

With this new table the parsing of the string **ababbb** is straightforward:

$$\begin{array}{l}
\texttt{\#} \lessdot \texttt{a} \lessdot \texttt{b} \lessdot \texttt{a} \lessdot \texttt{b} \lessdot \texttt{b} \gtrdot \texttt{\#} \\
\texttt{\#} \lessdot \texttt{a} \lessdot \texttt{b} \lessdot \texttt{a} \lessdot \texttt{b} \gtrdot \texttt{B} \gtrdot \texttt{\#} \\
\texttt{\#} \lessdot \texttt{a} \lessdot \texttt{b} \lessdot \texttt{a} \gtrdot \texttt{B} \gtrdot \texttt{B} \gtrdot \texttt{\#} \\
\texttt{\#} \lessdot \texttt{a} \lessdot \texttt{b} \doteq \texttt{S} \doteq \texttt{B} \gtrdot \texttt{B} \gtrdot \texttt{\#} \\
\texttt{\#} \lessdot \texttt{a} \doteq \texttt{S} \doteq \texttt{B} \gtrdot \texttt{\#} \\
\texttt{\#} \doteq \texttt{S} \doteq \texttt{\#}
\end{array}$$

The above total precedence table was constructed by ad hoc reasoning and handwaving, but we need an algorithm to implement this technique on a computer. It turns out that a grammar can have zero, one or several total precedence tables, and the problem is how to find one if it exists. Colmerauer [191] gives a set of equations a total precedence table for a given grammar has to obey, and provides several methods to solve these equations, including one that can reasonably be done by hand. But the procedures are lengthy and we refer the interested reader to Colmerauer's paper.

10.2.2 NSLR(1)

Postponing decisions in a total precedence parser was easy: just ignore the problem, read on and come back later to repair the damage. It is not that easy in an LR parser;

the reason is that an LR parser bets so heavily on the first handle that it finds it difficult to switch to an alternative set of hypotheses when the first set leads to problems. A good example is supplied by the structure of declarations in some (Pascal-like) programming languages:

```
VAR i, j, k: INT;
VAR x, y, z: REAL;
```

which can be described by the grammar

```
declaration_S  ->  VAR intvar_list ':' INT ';'
                 | VAR realvar_list ':' REAL ';'
 intvar_list   ->  intvar ',' intvar_list | intvar
      intvar   ->  variable_name
realvar_list   ->  realvar ',' realvar_list | realvar
     realvar   ->  variable_name
```

The reason we want exactly this grammar is that it allows us to attach semantics to the rules **intvar→variable_name** and **realvar→variable_name** that identifies the variable name with its proper type. But the Pascal-like syntax does not supply that information until the end of the declarations, which is why canonical LR techniques are not enough.

Since the long names in the above grammar are unwieldy in items and some of the tokens serve only to improve program readability, we shall use the abbreviated and abstracted grammar from Figure 10.12. The above declarations then correspond to **vvvi** and **vvvr**.

```
S_S  ->  I i
 S   ->  R r
 I   ->  V I
 I   ->  V
 V   ->  v
 R   ->  W R
 R   ->  W
 W   ->  v
```

Fig. 10.12. An abstract grammar of variable declarations

We will now show how to construct a new set of hypotheses for an SLR(1) parser when we have to abandon the original search for the first handle (Tai [197]). The initial state 1 of the SLR(1) parser for this grammar is

```
S→•Ii
S→•Rr
I→•VI
I→•V
V→•v
R→•WR
R→•W
W→•v
```

and the moment we shift over the first **v** we run into a reduce/reduce conflict in state 2 when we add the FOLLOW sets of **V** and **W** as look-aheads to the reduce items, in accordance with the recipe for SLR(1) parsers (Section 9.8):

```
V→v• [vi]
W→v• [vr]
```

The rest of the SLR(1) automaton is free of conflicts, but this reduce/reduce conflict is bad enough, since it reflects our inability to do the proper reduction of **v** until we have seen either the **i** or the **r**. But that token can be arbitrarily far away.

10.2.2.1 Creating New Look-Aheads

Clearly the present look-aheads are inadequate, so two questions arise: what look-ahead symbols do we use instead, and how do we obtain them. The look-aheads in SLR parsing derive from FOLLOW sets, which normally contain terminals only, since whatever comes after the end of the handle is the untouched input. In *non-canonical SLR(1)* (*NSLR(1)*) we try to obtain a 1-symbol (terminal or non-terminal) look-ahead by fully reducing a segment of the input that follows the item immediately. To determine what symbols qualify for this task, we need to know what fully reduced symbols can follow a given non-terminal A. This is easier than it sounds, since fully reduced symbols are exactly the symbols as they appear in the grammar. The set of fully reduced symbols that can follow a given non-terminal A is called $FOLLOW_{\mathrm{LM}}(A)$, since it is the same set of symbols which can follow A in sentential forms during leftmost production; hence the subscript LM.

The $\mathrm{FOLLOW}_{\mathrm{LM}}$ set can be obtained by running a variant of the FOLLOW set construction algorithm of page 245, in which the second step is replaced by (and simplified to!)

- We process all right-hand sides, including the S**#** one. Whenever a right-hand side contains a non-terminal, as in $A \rightarrow \cdots BX \cdots$, where X is a terminal or a non-terminal, we add X to $\mathrm{FOLLOW}_{\mathrm{LM}}(B)$. In addition, if $X \cdots$ derives ε, we add all symbols from $\mathrm{FOLLOW}_{\mathrm{LM}}(A)$ to $\mathrm{FOLLOW}_{\mathrm{LM}}(B)$.

This fills $\mathrm{FOLLOW}_{\mathrm{LM}}(A)$ with all unexpanded (= fully reduced) symbols that can follow A. For **V** this yields {**I**,**i**}, where the **I** comes directly from **I→VI**, and the **i** comes from **I→V** and **S→Ii** through the ε clause in the algorithm. Likewise, $\mathrm{FOLLOW}_{\mathrm{LM}}$(**W**) = {**R**,**r**}. Note that $\mathrm{FOLLOW}_{\mathrm{LM}}$ is in general neither a subset nor a superset of FOLLOW.

Now that we have determined the new look-aheads we can turn to the problem of obtaining them from the rest of the input. Actually the new non-terminal look-aheads can be seen as new hypotheses for finding the handle; only now the handle will be found (if possible) in the first segment of the rest of the input. So we add the items for •**I** and •**R** to state 2, plus all the prediction items that are brought in by these:

```
V→v• [Ii]
W→v• [Rr]
I→•VI
I→•V
V→•v
R→•WR
R→•W
W→•v
```

If the state still has a conflict even when using $\text{FOLLOW}_{\text{LM}}$ rather than FOLLOW, the grammar is not suitable for this technique (but see the next section).

The new items may cause transitions to new, non-canonical states that were not present in the original SLR(1) parser. These states are used by the parser when it hunts for a non-first handle. Of course these non-canonical states can again have conflicts, and if they cannot be solved by the same technique, the grammar is again not NSLR(1).

10.2.2.2 Finding Minimum Look-Ahead Sets

The above state is not yet a proper NSLR(1) state but rather a LSLR(1) state, for *L*eftmost SLR(1), since it is based on $\text{FOLLOW}_{\text{LM}}$. The LSLR technique will work for the grammar of Figure 10.12, but it can be shown that the requirement for "fully reduced" items is overly strong. Quite often a less reduced look-ahead will do, and by using such a look-ahead we can occasionally avoid a non-canonical state which would have had a conflict.

The minimum set of look-ahead symbols can be found as follows (Tai [197]). We first determine the first symbols each look-ahead non-terminal X of a reduce item $A \rightarrow \cdots [\cdots X \cdots]$ goes through on its way to being fully reduced. These are easily found, since they are the symbols right after the dot in the prediction items resulting from X. For `I` they are {`V`,`v`} and for `R` we get {`W`,`v`}. We tentatively add these to the reduce look-aheads, resulting in

```
V→v• [IVvi]
W→v• [RWvr]
```

We see that we have now created a reduce/reduce conflict, but that does not surprise us since we knew already that we had to reduce the `v` to something, so we remove the `v`. The whole state 2 now becomes

```
V→v• [IVi]
W→v• [RWr]
I→•VI
I→•V
V→•v
R→•WR
R→•W
W→•v
```

but this still has shift/reduce conflicts. Remarkably, these are easily removed: when we have a choice between using, say, `V` to resolve the conflict between the first two

items and shifting over **V** to find another non-terminal, **I**, which will then just later serve to resolve the same conflict, we of course choose to reduce and not to shift. So we can remove the shift items that cause shift/reduce conflicts (but only those that were added to resolve the original conflict)! Note that this differs from the traditional preference for a shift on a shift/reduce conflict presented in Section 9.9. The item set has become a lot smaller now:

```
V→v• [IVi]
W→v• [RWr]
V→•v
W→•v
```

And since **I** and **R** no longer appear on the left-hand side of any item, they will not pop up as look-aheads and can be removed:

```
V→v• [Vi]
W→v• [Wr]
V→•v
W→•v
```

This is the final form of the NSLR(1) state 2.

Tai [197] proves that applying this procedure to a conflict-free LSLR(1) state cannot cause the resulting NSLR(1) state to have conflicts, but the proof is lengthy. In other words, there are no LSLR(1) grammars that are not also NSLR(1). There exist, however, grammars that are NSLR(1) but not LSLR(1); this is caused by states in the LSLR(1) parser that are absent from the NSLR(1) parser. For examples see Tai's paper.

The complete NSLR(1) automaton is shown in Figure 10.13. The other reduce states (5, 6, 7, 10, 11, 12) have not been subjected to the SLR-to-NSLR transformation, since they are already adequate SLR(1) states.

10.2.2.3 A Complete NSLR(1) Parsing Example

The input string **vvi** is now parsed as follows:

①	v v i #	
① v ②	v i #	
① v ② v ②	i #	reduce by **V→v**
① v ② V	i #	

Here a new look-ahead **V** is obtained in state ②, which causes a further reduce; one way of understanding this is by pushing the **V** back into the input stream:

① v ②	V i #	reduce by **V→v**
① V	V i #	

and push back again, followed by two shifts:

①	V V i #
① V ③	V i #
① V ③ V ③	i #

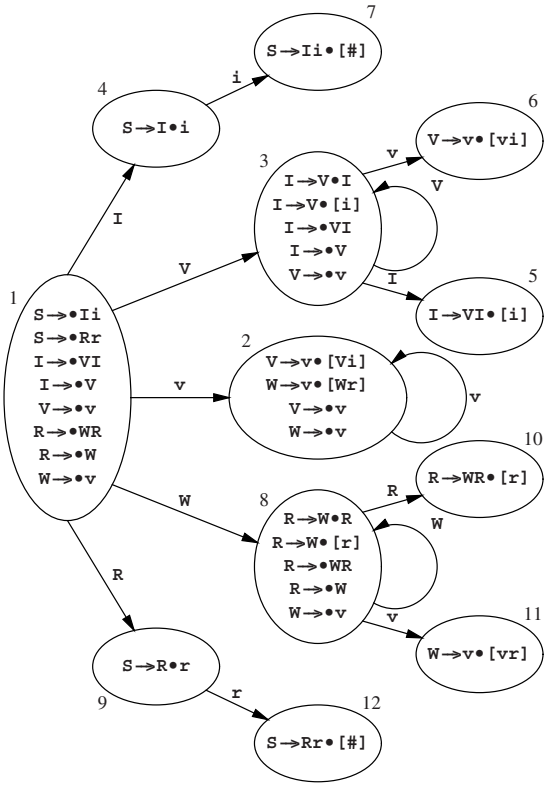


Fig. 10.13. NSLR(1) automaton for the grammar in Figure 10.12

The look-ahead `i` in state ③ asks for a reduce by `I→V`, then by `I→VI` in state ⑤, and then on to the start symbol `S`:

① V ③	I i #
① V ③ I ⑤	i #
①	I i #
① I ④	i #
① I ④ i ⑦	#
① S	#

It is interesting to note that states 6 and 11 cannot be reached...

Salomon and Cormack [200] give an explicit algorithm for NSLR(1), and apply it to complicated problems in programming language parsing.

10.2.3 LR(k,∞)

As we have seen in Sections 9.6 and 9.8, the essential difference between SLR(1) parsing and LR(1) parsing is that an SLR(1) parser uses the FOLLOW set of a nonterminal A as the look-ahead set of an item $A \rightarrow \cdots$, whereas an LR(1) parser con-

structs the look-ahead set precisely, based on the look-ahead(s) of the item or items $A \rightarrow \cdots$ derives from.

When in a non-canonical SLR(1) parser the necessity arises to create new items because we abandoned a reduce item $A \rightarrow \alpha\bullet$, we can do so relatively easily by expanding the non-terminals in $\text{FOLLOW}_{\text{LM}}(A)$ and then do some clean-up. And when we then are forced to abandon these new items, we can again turn to the $\text{FOLLOW}_{\text{LM}}(A)$ set to obtain new non-terminals and from them new items. This is because the look-ahead of an item in an SLR(1) parser does not depend on its origin, but only on A.

In a non-canonical LR parser we have to collect much more information about the set of tokens that can follow a given item I in a given item set (state). First its construction should follow the principles of LR parsing, which means that it should derive from the look-ahead information of the items I derives from. And second, the information should cover the entire rest of the input, since we do not know how often and until what point we need to postpone our decisions. Non-canonical LR parsing was first proposed by Knuth [52], but it was Szymanski [194] who gave an algorithm for its construction. The algorithm yields an *LR(k,∞)* parser, where k is the length of the look-ahead and ∞ (infinity) is roughly the number of times a decision can be postponed (see Section 10.2.3.4 for more on this subject).

The regular right context grammar explained in Section 9.12.2 suggests itself as a good representation of the look-ahead information required by a non-canonical LR parser, but it is not good enough. The regular grammar describes all possible right contexts of an item I that can occur, over all paths along which a state can be reached, but in an actual parsing we know that path precisely. It is easy to see that in the grammar `S→(S)|a` the regular right context of the item `S→a•` is `)`*, but when in a parsing we have seen the first part of the input `(((a` we know that the exact right context is `))))`. And it is this kind of exact right context that we want to use as a look-ahead in LR(k,∞) parsing; it is a subset of the regular right context grammar and has to be constructed during parsing.

So for the moment we have two problems: how to derive the LR(k,∞) right contexts and how to use them during parsing.

10.2.3.1 An LR(1,∞) Parsing Example

We will again use the abstract grammar for declarations in Figure 10.12 on page 360 we used in explaining NSLR(1) parsing and stick to one-token look-ahead, so $k = 1$. Since we want to see exactly what happens to the look-aheads, we will build up the states very careful. The kernel items of the initial state, 1_k of the LR(1,∞) parser are

```
S→•Ii #
S→•Rr #
```

Expanding the non-terminals after the dot yields the expanded initial state, 1_e,

```
S→•Ii #
S→•Rr #
I→•VI i#
I→•V i#
V→•v Ii#
V→•v i#
R→•WR r#
R→•W r#
W→•v Rr#
W→•v r#
```

which differs from the initial state of the NSLR(1) parser (page 360) only in that the full right context is kept with each item. For example, the right context of **I→•VI** is **i#** because the item derives from **S→•Ii #**.

Suppose the input is **vvi**. As all LR parsers, the LR(1,∞) parser starts with an empty stack and the input concatenated with the end marker as "rest of input" (Figure 9.12, page 279). We will write this configuration as **•vvi#**, where • is the gap. The look-ahead in this configuration is a **v**. Rather than examining each item to see how it reacts to this look-ahead, we first simplify the state by removing all items that do not have **v** as their dot look-ahead and then see what the rest says. The filtered state 1_f is

```
V→•v Ii#
V→•v i#
W→•v Rr#
W→•v r#
```

All items agree on the action: shift, which yields state 2_k:

```
V→v• Ii#
V→v• i#
W→v• Rr#
W→v• r#
```

and the configuration changes to **v•vi#**. The state suggests two different reduce operations, so we need look-ahead, which we obtain by expanding the dot look-aheads (**I** and **R**). In canonical LR parsing they are replaced by their FIRST sets (FIRST(**Ii#**) and FIRST(**Rr#**), respectively) but here we want their FIRST sets plus their expansions since one of these may be the basis for a non-canonical reduce operation further on. This causes dotted items in the look-ahead parts of other items, a strange but useful construction. Szymanski does not give them a name, but we will call them "dotted look-aheads"; and we will call the non-look-ahead item the "active item".

When we expand the **I** in **V→v• Ii#** to **VI**, we obtain the item

```
V→v• [I→•VI]i#
```

where **V→v•** is the active item, **[I→•VI]** is a dotted look-ahead representing the first part of the right context, and **i#** is the rest of that context. This indicates that the complete right context is **VIi#**, with the understanding that when the **VI** gets

recognized, it must be reduced to **I**. Szymanski uses the notation **]**$_5$**VI]**$_3$**i#** for this item, where the subscripted bracket **]**$_n$ means: "when you get here you can reduce by rule number n". This is more compact and more efficient algorithm-wise but less informative. Note that for $k > 1$ there can be more than one dotted look-ahead in an item.

Applying this expansion to all items in state 2_k we obtain state 2_e:

```
V→v• Ii#
V→v• i#
W→v• Rr#
W→v• r#
V→v• [I→•VI]i#
V→v• [I→•V]i#
V→v• [V→•v]Ii#
V→v• [V→•v]i#
W→v• [R→•WR]r#
W→v• [R→•W]r#
W→v• [W→•v]Rr#
W→v• [W→•v]r#
```

Now we can filter out the items that are compatible with the look-ahead **v**, yielding state 2_f:

```
V→v• [V→•v]Ii#
V→v• [V→•v]i#
W→v• [W→•v]Rr#
W→v• [W→•v]r#
```

We see that there is still no agreement among the items, so we give up on the hypothesized reduces **V→v•** and **W→v•**, and promote the dotted look-aheads to active items:

```
V→•v Ii#
V→•v i#
W→•v Rr#
W→•v r#
```

Now we shift; this shift is certain to succeed, since we have just made sure all items had a **v** as a the dot look-ahead. The result is

```
V→v• Ii#
V→v• i#
W→v• Rr#
W→v• r#
```

which brings us back to state 2_k and changes the configuration to **vv•i#**. It may seem natural that we come back to state 2_k here, because we have read just another **v**, but it isn't. If the rule **I→VI** had been **I→VIx**, the first item in state 2_k had been **V→v• Ixi#** and that of the above state **V→v• Ixxi#**. This shows the profound effect of keeping the exact entire right context.

Expanding state 2_k yields again state 2_e, but now the look-ahead is **i**! Filtering it with this look-ahead yields the state

```
V→v• i#
```

and now a unanimous decision can be taken. We reduce **v** to **V** and the configuration becomes **v•Vi#**. After the first **v** the parser was in the state 2_e, and this state must now be filtered with look-ahead **V**. This yields another reduce state:

```
V→v• [I→•VI]i#
V→v• [I→•V]i#
```

We reduce the first **v** to **V**, with the new configuration **•VVi#**, which provides a look-ahead **V**, with which we filter state 1_e, etc. The rest of the parsing proceeds similarly.

10.2.3.2 The LR(k,∞) Algorithm

The basic loop of a non-canonical LR(k,∞) parser is different and more complicated than that of a canonical LR parser:

- Consider the item set p_k on the top of the stack.
- Expand the look-aheads in each of the items if they are non-terminals; they then yield dotted look-aheads. This results in a state p_e.
- Filter from p_e the items that have the actual look-ahead as their dot look-ahead. This results in a state p_f.
- See if the items in p_f lead to a decision. Five decisions are possible: reduce; accept; reject the input; reject the grammar as not LR(k,∞); and reject the grammar as ambiguous. They are covered in detail below.
- If the items in p_f do not lead to a decision, shift, as described below. The shifted token and the new item set resulting from the shift are stacked.

We shall now examine the five possible decisions in more detail.

- Since it is the purpose of LR parsers to produce one single parse tree, each reduction we do must be correct, so we reduce only if all items in p_f appoint the same reduction.
- We accept the input when it has been reduced to the start symbol. The look-ahead together with the end marker make sure that this can only happen at the end of the input.
- We reject the input as erroneous when there are no items left in the top state. When that happens there are no possibilities left for the right context, so no further input can ever complete the parse tree.
- If we have abandoned the active item it is possible that there is no dotted look-ahead left to eventually turn into an active item. Now we created dotted look-aheads hoping that after a while one of them would be recognized and reduced to a single non-terminal, which would be used as a look-ahead to resolve an earlier LR conflict — but if there is no dotted look-ahead that will never happen, and the LR(k,∞) method is not strong enough to handle the grammar.

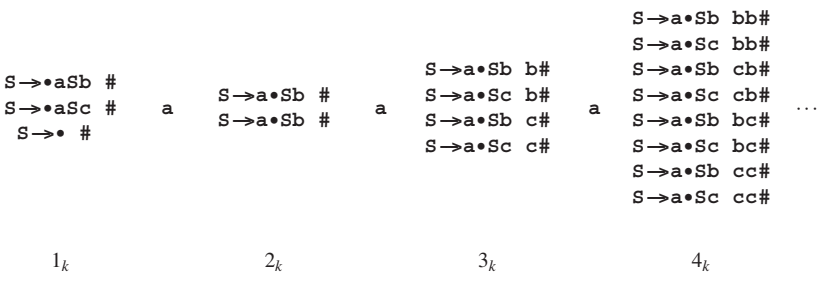
- It is possible that when we reach the end of the input we still have more than one item left, which means that we still have not been able to make a decision, and still more than one completion of the parse tree is possible. So the input is ambiguous, the grammar is ambiguous, and the grammar is not LR(k,∞).

Normally in an LR parser, when we shift we move the dot one place to the right in the (active) item, and when the dot happens to be at the end of the item we would not shift but rather reduce or have a conflict. Here we may need to shift even if the active item is a reduce item, and we have seen above how we do that: we abandon the active reduce item. If there is a dotted look-ahead at the front of the right context now, it is promoted to active item with the dot at the beginning. And if there is not, the item will continue for a while without active item, until a dotted look-ahead is finally shifted to the front. This can only happen for $k > 1$.

LR(k,∞) parsing is much more powerful than LR(k) parsing, but this power comes at a price. It is undecidable whether a grammar is LR(k,∞), and we have seen above that even during parsing we can find that the grammar is not LR(k,∞) or is even ambiguous. So we can successfully parse millions of strings with a grammar and only then find out that it was not LR(k,∞) and the parser was unsound. Also, the method has several serious implementation problems (see next section), but then again, it *is* the strongest linear-time parsing technique for unambiguous grammars known.

10.2.3.3 Problems with and Fixes for the LR(k,∞) Parser

We just claimed that LR(k,∞) parsers have linear time requirements, but the signs are not favorable. Suppose we have a grammar **S→aSb | aSc | ε** and an input **aaaa**⋯. Then we meet the following kernel item sets:



etc., so we see that the size of the item set grows exponentially. And when we try the algorithm on the left-recursive grammar of Figure 9.14, even the initial state is infinitely large because it contains infinite sequences like

```
E→•E-T $#
E→•E-T -T$#
E→•E-T -T-T$#
E→•E-T -T-T-T$#
E→•E-T -T-T-T-T$#
E→•E-T -T-T-T-T-T$#
E→•E-T -T-T-T-T-T-T$#
⋮
```

The cause of these problems is that right contexts are finite-state (Type 3) languages and the above algorithm constructs finite-choice (Type 4) expressions for them. The ever-growing states 2_k, 3_k, ... above actually contain only two items regardless of the number of **a**s read; for example, state 4_k is actually

```
S→a•Sb [bc][bc]#
S→a•Sc [bc][bc]#
```

and the infinite sequence can be condensed into a single regular expression:

```
E→•E-T (-T)*$#
```

So the expansion step in the LR(k,∞) algorithm must be extended with grammar-to-expression transformations like those in Figure 5.19. Unfortunately the details of this step have not been published, as far as we know.

With these transformations the item sets no longer grow infinitely or exponentially, but they still grow, linearly. After having processed seven **a**s from the input the two-item state is

```
S→a•Sb [bc][bc][bc][bc][bc][bc]#
S→a•Sc [bc][bc][bc][bc][bc][bc]#
```

This can*not* be condensed to

```
S→a•Sb [bc]*#
S→a•Sc [bc]*#
```

because only exactly seven **b**s or **c** are acceptable, one from the active item and six from the look-aheads. Since the look-ahead sets are copied from item to item, a linear growth in look-ahead size translates into a quadratic time requirement. Fortunately there is a simple way to fix this problem. New look-aheads are created only when a non-terminal is expanded; during this expansion an item $A \rightarrow \alpha \bullet B \beta\ \gamma$ causes an item $B \rightarrow \bullet\delta\ \beta\gamma$ to be created, so the look-ahead changes from γ to $\beta\gamma$: the addition is always at the front of the old look-ahead. So we can implement the item $B \rightarrow \bullet\delta\ \beta\gamma$ as $B \rightarrow \bullet\delta\ \beta\mathcal{P}$ where $\mathcal{P}$ is a pointer to γ, the look-ahead of the parent item. This reduces the copying of an item to constant costs, and the overall parse time requirements to linear in the length of the input. (The dotted look-aheads complicate the algorithm somewhat but can be handled in basically the same way.)

LR(k,∞) parsing is the most powerful linear-time parsing algorithm known today. It can handle many more grammars than LR(k) but it cannot handle all unambiguous grammars; an example of an unambiguous non-LR(k,∞) grammar is $S \rightarrow aSa|\varepsilon$. It is

undecidable whether a grammar is LR(k,∞), and the parser discovers its own insufficiency only while parsing a string that hits one of its weak spots. LR(k,∞) is also called *NLR(k)*, for Non-canonical LR(k).

10.2.3.4 LR(k,t)

LR(k,∞) parsing may be the most powerful linear-time parsing algorithm known today, but it has one problem: decidability. Not only does the undecidability surround it with a blanket of uncertainty, it also prevents the creation of a table-driven version. Section 9.5 has shown us how much less convenient and efficient the interpretive parser in Figure 9.16 is compared to a table-driven one based on the deterministic automaton of Figure 9.17, and we would like to "upgrade" our LR(k,∞) parser in a similar way. One reason why we cannot is that the LR(k,∞) parser has an infinite number of states, as the examples in the previous sections show. If it had a finite number of states, we could construct them all, and thus achieve decidability and a table-driven version at the same time.

So it becomes interesting to see why there are infinitely many states. There are only a finite number of dotted items, and a much larger but still finite number of combinations of them, but it is the unbounded length of the right contexts that causes the number of states to be infinite. This raises the question why we need unbounded length right contexts, especially if we use a finite look-ahead of k tokens only. The answer is that the segment of the right context after the first k tokens serves one important purpose: to create dotted look-aheads which turn into active items when the original active item is abandoned. So intuitively it should help if we restricted the number of times the active item in a given item can be abandoned to say t; this leads to *LR(k,t)* parsing. (This notation also explains the name LR(k,∞).)

To understand that this works we go back to the explicit, finite choice implementation of right contexts, where a right context is just a string of terminals and non-terminals. Now suppose we have an LR(1,2) item `P→p• RSTuv#` resulting from a shift over `p` in a grammar which contains the rules `P→p`, `R→r`, `S→s`, and `T→t`, among many others, and we follow this item through the shift and expand actions performed on it. We will assume that at each decision point there are other, conflicting, items in the same state which force us to abandon the active item; this assumption causes the maximum utilization of the right context.

Since $t = 2$, we can abandon the active item only twice, and to keep track of this we record the number of abandons with the item; see Figure 10.14. In step 2 of the table we expand the look-ahead `R` to a dotted look-ahead, which turns into an active item in step 3, due to giving up on `P→p•`. Also the counter rises to 1. Similar actions bring us to step 6 where the counter has risen to 2, and no further "abandon and shift" is possible.

Exhausting the number of abandons allowed means two things. The first is that when the exhausted item occurs during parsing and we still cannot make a decision, the grammar is not LR(1,2) and the parsing fails. The second is more important for our purposes: we see that the trailing `uv#` never played a role, so we can remove them from the original item `P→p• RSTuv#`, truncating it to `P→p• RST`.

Step	Action	Resulting item
1	shift over **p**	**P→p• RSTuv#** 0
2	expand	**P→p• [R→•r] STuv#** 0
3	abandon and shift over **r**	**R→r• STuv#** 1
4	expand	**R→r• [S→•s] Tuv#** 1
5	abandon and shift over **s**	**S→s• Tuv#** 2
6	expand	**S→s• [T→•t] uv#** 2
7	stop	

Fig. 10.14. Development of an LR(1,2) item until exhausted

In this way we can for each item find out how much of its right context is needed to allow at most t abandons. This keeps the right contexts limited in length and keeps the number of possible LR(k,t) states finite, so we can construct a table-driven parser. Also, we do not have to wait until parse time to find conflicts; they reveal themselves during table generation, as with LR parsers: we have achieved decidability! Szymanski [194] gives details.

Just as the power of LR(k,∞) came at a price, decidability, the decidability of LR(k,t) comes at a price: power. Although LR(k,t) can handle many more grammars than LR(k), it cannot handle more languages.

10.2.3.5 Discussion

LR(k,∞) parsing is the strongest linear-time parsing algorithm known today, both with respect to grammars and to languages. Suitability of a given grammar cannot be checked in advance, so the parser may reject the grammar while parsing. The full algorithm is quite complicated and carries a heavy performance penalty, as states, look-aheads and right contexts are constructed on the fly.

LR(k,t) parsing is the strongest decidable linear-time parsing algorithm known today, with respect to grammars. It handles many more grammars than LR(k), but can handle deterministic languages only. Its table-driven implementation is as efficient as LALR(1) parsing, but the table construction algorithm is very complex and the tables can be large.

Hutton [202] describes non-canonical LALR(k) (NLALR(k)) and (NLALR(k,t)), also called LALR(k,t). It turns out that it is undecidable if a grammar is NLALR(k), but it is decidable if a grammar is NLALR(k,t), just as with LR(k,∞) and LR(k,t).

As Szymanski [194] and Farré and Fortes Gálvez [207] point out, non-canonical LR parsing does LR parsing with context-free look-ahead. It could with some justification be called LR-context-free, in analogy to LR-regular.

10.2.4 Partitioned LR

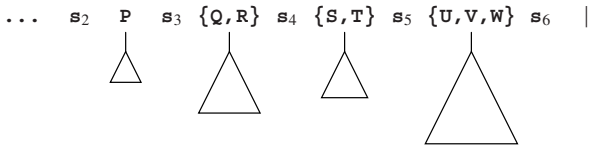
When an LL(1) parser is confronted with two alternatives, both starting with the same token, as in

```
A  →  P | Q
P  →  a
Q  →  a
```

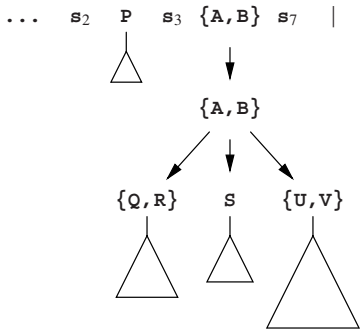
it has a FIRST-FIRST conflict; but an LR parser merrily shifts the **a** and accepts both alternatives, leading to a state {**P→a•,Q→a•**} (which is why there is no such thing as a shift-shift conflict). When an LR parser is confronted with two possible reduction rules, as in the state {**P→a•,Q→a•**}, it has a reduce/reduce conflict; but a *Partitioned LR* parser merrily reduces the **a** to both non-terminals, resulting in a set {**P**,**Q**}. This is of course only possible when all right-hand sides in the reduction have the same length.

10.2.4.1 Sets of Non-Terminals as LR Stack Entries

In a Partitioned LR parser, the LR stack can contain sets of non-terminals in addition to the usual non-terminals. The uncertainty that that implies can often be resolved by later reductions, as the following example shows. Suppose the top few elements of the stack are



where the non-terminal sets are linked to partial parse trees as the non-terminals did in Figure 9.12; so {**U**,**V**,**W**} points to a tree that can represent a terminal production of a **U**, a **V** or a **W**, and similarly for the other sets. Now suppose the top state s_6, possibly with help from some look-ahead, indicates that the parser should reduce using the rules **A→QSU** and **B→RSV**. Then the top three non-terminal sets get scooped up from the stack and linked to a single node for {**A**,**B**}:

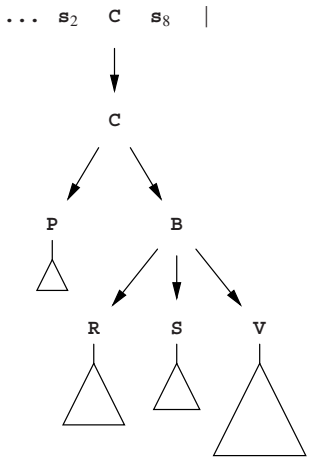


We see that the set {**U**,**V**,**W**} has been narrowed down to {**U**,**V**} in the process, and that the second member of the right-hand side has been fixed to **S**, since the **W** and

the **T** are not compatible with the right-hand sides of the reduction rules **A→QSU** and **B→RSV**.

We also note that, unlike canonical LR states, Partitioned LR states can contain reduce items with unrelated right-hand sides. In a canonical LR state each right-hand side must be a suffix of another right-hand side or vice versa, for example **F→AbC•** and **G→bC•**, since both must match the top of the stack ··· **AbC•**. Actually the same is true in Partitioned LR parsers, but since the stack contains sets of non-terminals, the right-hand sides of rules in a state have much more leeway, and indeed the reduce items **A→QSU•** and **B→RSV•** both match the top of the stack above. If they did not they would not have survived the shifts over {**Q**,**R**}, {**S**,**T**} and {**U**,**V**,**W**}.

Now suppose state s_7 tells us to reduce with the rule **C→PB**. The reduction refines the {**A**,**B**} to a single **B**. This information then propagates into the tree for {**A**,**B**}, fixing {**Q**,**R**} to **R** and {**U**,**V**} to **V**, resolving all uncertainties:



We see that a reduce action in a Partitioned LR parser entails updating the parse tree in addition to the usual task of creating a new parse tree node. Actually one can distinguish these two tasks even in a canonical LR parser: first the node is created with its children, and then the node is labeled with the proper non-terminal. Similarly, we have seen in Section 9.2.2 that operator-precedence parsers construct skeleton parse trees: nodes are just constructed; they never get labeled. Node construction and node labeling are two fairly independent actions; only in canonical LR parsing do they occur simultaneously.

10.2.4.2 A Partitioned LR Parsing Example

It is relatively easy to construct a Partitioned LR handle-finding automaton, and we even have the choice between LR(0), SLR(1), etc. for the look-ahead. We first construct the canonical LR automaton, of the desired kind. When it has no conflicts, we are of course done and do not need to resort to non-canonical techniques. When it

has shift/reduce conflicts or reduce/reduce conflicts with rules of unequal lengths, the Partitioned LR technique cannot help us (but see Problem 10.17). But when there is a state with a reduce/reduce conflict with rules of equal lengths, we mark the state for reduce with all these rules for the appropriate look-aheads. This creates a set of non-terminals $A = \{A_1, A_2, \ldots, A_n\}$ as a potential stack entry. We examine each state that contains at least one item $B \rightarrow \cdots \bullet A_k \cdots$ and see how it reacts to shifting over the set A. This may create new states, and so on, but the process will eventually terminate. If this removes all conflicts, the grammar is Partitioned LR of the desired kind.

Figure 10.15 shows part of the Partitioned SLR(1) automaton for the grammar of Figure 10.12, and we will use it to demonstrate the parsing of the input **vvvi**. The

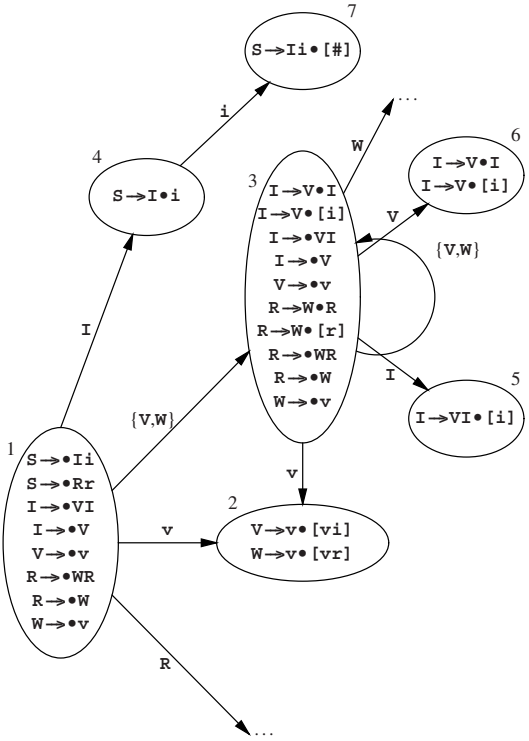


Fig. 10.15. Part of a Partitioned SLR(1) automaton for the grammar of Figure 10.12

initial state ① is that of an SLR(1) parser, as it was in Figure 10.13. Shifting over the **v** brings us to state ②, which under a look-ahead of **v** asks for a reduce to {**V**,**W**}:

① **v** ② **v v i #** reduce **{V,W}→v**

Next, we shift over the {**V**,**W**} and the second **v**:

① **{V,W}** ③ **v** ② **v i #** reduce **{V,W}→v**

and history repeats itself. But after the third **v** is shifted, state ② finds a look-ahead **i** and can now authorize a reduce to **V**:

① {V,W} ③ {V,W} ③ v ②	i #	reduce V→v

Shifting over the **V** brings us to state ⑥, which reduces the **V** to an **I**, which results in state ⑤ on the top of the stack. This causes the reduction of {**V**,**W**}**I** to **I**, which requires fixing the {**V**,**W**} to **V**. From there the road to the end state is clear:

① {V,W} ③ {V,W} ③ V ⑥	i #	reduce I→V
① {V,W} ③ {V,W} ③ I ⑤	i #	reduce I→VI, refining {V,W} to V
① {V,W} ③ I ⑤	i #	reduce I→VI, refining {V,W} to V
① I ④ i ⑦	#	reduce S→Ii

There are striking similarities but also considerable differences between this Partitioned LR example and the NSLR(1) parsing example on page 363. In Partitioned LR non-terminals do not figure as look-ahead, and the shift and reduce actions are more similar to those of an LR parser than of an NSLR parser. On the other hand an NSLR parser does not need to update the parse tree.

10.2.4.3 Discussion

A restricted parser based on the above principles was described by Madhavan et al. [206]. The described parser is used as a structuring tool in compiler design in a technique called Graham–Glanville code generation.[1] It requires grammars to use only two types of rules, $A \rightarrow B_1 \cdots B_n t$ and $A \rightarrow B$, with the restriction that if the terminal t occurs in more than one rule, all these rules must have the same value for n. This requirement ensures that the grammar is Partitioned LR(0), but makes the parser impossible to use in a more general setting. No other publication on Partitioned LR is known to us.

One practical advantage of Partitioned LR is that it delivers a partially resolved parse tree, which can then be disambiguated on the fly or off-line by grammatical or external means. This is exploited by Madhavan et al. by incorporating a cost function in the parser; this cost function cooperates with the parser to find the optimal structuring of the input as to costs. For details see Madhavan et al. [206].

It is easy to see that the class of Partitioned LR grammars and that of NSLR grammars are incommensurable. NSLR can handle grammars with reduce-reduce conflicts with rules of unequal length (for example, the grammar of Figure 10.12 with **W→v** replaced by **W→vv**) which Partitioned LR cannot. Partitioned LR can handle some ambiguous grammars (for example, the grammar of Figure 10.12 with **S→Rr** replaced by **S→Ri**), which NSLR cannot. In fact, the ability to handle ambiguous

[1] In Graham–Glanville code generation a bottom-up parser is used to structure the stream of intermediate machine instructions originating from the intermediate code generator in a compiler into final machine instructions, which are specified to the parser as grammar rules.

grammars is one of the strong points of Partitioned LR, since it allows the efficient construction of ambiguous parse trees, which can then be disambiguated on external criteria.

It is clear that Partitioned LR parsing needs more research.

10.3 General Non-Canonical Parsing

In a sentence like "The neighbors invited us to a barbecue party", the word that carries the most syntactic and semantic information is "invited". It tells us many things: the action is in the past; it is a transitive verb so we should be looking for two noun phrases, one for the subject and one for the object; and if we have a good data base entry for "to invite" we know that the subject and object are very likely human, and that there might be a preposition phrase starting with "to". With this knowledge we can identify the noun phrase "the neighbors" as the subject, "us" as the object, both duly human, and "to a barbecue party" as the preposition phrase. In the noun phrase "the neighbors", the most significant word is "neighbors"; in "to a barbecue party" it is "party"; etc. And we already see a parse tree emerging.

When we want to develop this idea into a parsing technique, we meet two problems: how do we tell the computer what is the most important component of a phrase; and how does the computer find that component in the input. The answers are "head grammars" and "head-corner parsing," respectively. A *head grammar* is a CF grammar in which one member in each right-hand side is marked as the "head" of that right-hand side. The top level of a very simple head grammar of English could look like this:

```
  S_S  →  NP VP NP PP | ···
   NP  →  ART? NOUN | PRON | ···
 NOUN  →  ··· | 'neighbors' | ···
 PRON  →  ··· | 'us' | ···
  ART  →  'the' | 'a'
   VP  →  ··· | 'invited' | ···
   PP  →  ··· | 'to' NP | ···
```

Here **NP** stands for "noun phrase", **VP** stands for "verb phrase", and **PP** for "preposition phrase". We use a bar over a symbol to indicate that it is the head; if there is only one symbol in a right hand side it is the head automatically. Head grammars were first discussed by Proudian and Pollard [198], and made popular by Kay [199].

Several algorithms have been published to exploit the head information and to lead the parser to the proper heads of the phrases. These are called *head-corner parsers*, for reasons to be explained below. They include modified chart parsers (for example Proudian and Pollard [198], or Kay [199]) or modified Earley parsers (for example Satta and Stock [201], or Nederhof and Satta [203]).

An intuitively appealing version of a head-corner chart parser is given by Sikkel and op den Akker [204]. We start from the start symbol *S*. In each of its right-hand sides we expand the symbol marked "head", unless it is a terminal, and at the same

time construct the corresponding partial parse tree. We then continue expanding non-terminals marked "head" in these partial parse trees, producing more and more partial parse trees, until in each of them we reach a terminal marked "head." This process yields a set of spines similar to the ones produced in left-corner parsing in Section 10.1.1.1 and Figure 10.3. Whereas left spines are constructed by systematically expanding the *leftmost symbol* until we meet a terminal, head spines are constructed by systematically expanding the *head symbol* until we meet a terminal. Like the left spines in Figure 10.4, head spines can contain cycles. In fact, if the head in every right-hand side in the grammar is the leftmost symbol, head-corner parsing turns into left-corner parsing. This is how head-corner parsing got its name, in spite of the fact that no corner is involved.

This preparatory step, which is independent of the input, yields a large number of head spines, each connecting S to some terminal t_A through a rule $S \rightarrow \alpha\bar{A}\beta$, where t_A is eventually produced by A through a chain of head non-terminals. For each t_A found in the input at position p, a partial parse tree P is now constructed from S to t_A. For such a parse tree P to be correct, α has to produce the segment $1 \ldots p-1$ of the input and β the segment $p+1 \ldots n$.

Now suppose α is actually BC. We then construct all head spines for B and C, and for all head spine terminals t_B produced by B and t_C produced by C that occur in that order in the input segment $1 \ldots p-1$ we connect their spines to P. Next we do the same for β and the segment $p+1 \ldots n$. If the required spines cannot be found P is discarded. We continue this process recursively on both sides, until all input tokens are accounted for. We have then constructed all possible parse trees. Sikkel and op den Akker [204] use chart parsing arcs to do the administration, and give many details in their paper.

The above explanation actually missed the point of head-corner parsing, which is that semantic considerations can be introduced to great profit at an early stage. When we have a head spine $S \rightarrow \cdots\bar{A}\cdots$, $A \rightarrow \cdots\bar{F}\cdots$, $F \rightarrow \cdots\bar{G}\cdots$, $G \rightarrow \cdots t_A\cdots$, we can take all the semantic information attached to t_A — its "attributes" — propagate them up the spine and use them to restrict possible head spines to be attached to the dots on the left and the right of the spine. Suppose, for example, that t_A is a verb form identifying a feminine plural subject, and suppose the dots to the left of $\bar{F}$ in the rule for A include a possible subject, then only head spines ending in a terminal which identifies a feminine plural form need to be considered for that position. This tends to quickly reduce the search space.

An in-depth description of a head-corner parser implemented in Prolog is given by van Noord [205].

We see that head-corner parsing identifies the nodes in the parse tree in a characteristic non-standard way. That and its close relationship to left-corner parsing has led the authors to classify it as a general (non-deterministic) non-canonical method and to cover it in this chapter; but we agree that the taxonomy is strained here.

10.4 Conclusion

Non-canonical parsing is based on postponing some decisions needed in canonical parsing, but postponement is something that is open to interpretation, of which there are many. Almost all decisions can be postponed in more than one way; a whole range of parsing techniques result, and the field is by no means exhausted.

The advantage of non-canonical parsing techniques is their power; LR(k,∞) is the most powerful linear parsing technique we have. Their main disadvantage is their sometimes inordinate complexity.

Problems

Problem 10.1: What conclusion can be drawn when a production chain automaton (like the one in Section 10.1.1.2(a)) happens to be deterministic?

Problem 10.2: Analyse the movements of the strong-LC(1) parser from Figure 10.6 on the incorrect input string `n)`. What nodes did it predict before detecting the error?

Problem 10.3: Why is it impossible for a left-corner parser to produce the analysis shown in Figure 10.7 immediately, including the proper parentheses?

Problem 10.4: Construct the full-LC(1) parse table corresponding to the strong-LC(1) one in Figure 10.6.

Problem 10.5: Rosenkrantz and Lewis, II [101] and Soisalon-Soininen and Ukkonen [104] use slightly different and incompatible definitions of LC(k). Map the differences.

Problem 10.6: *Project:* Implement an LC(1) parser for the example in Section 10.1.1 using an LL(1) parser generator and add code to produce a correct parse tree.

Problem 10.7: *Project:* The treatment of left-corner parsing is marred by an asymmetry between rules with right-hand sides that start with a terminal and those that start with a non-terminal. This nuisance can, in principle, be remedied by introducing a non-terminal $\mathcal{E}$ which produces only ε, and putting an $\mathcal{E}$ in front of all right-hand sides that start with a terminal symbol. Rethink the examples and algorithms of Section 10.1.1 for a thus modified grammar.

Problem 10.8: By definition non-canonical parsers take their parsing decisions in non-standard order. The parsers from Figures 6.17 and 10.10 print the rules involved in the parse tree in standard postfix order, just as any LR parser would. Is there a contradiction?

Problem 10.9: Arguably the simplest non-canonical bottom-up parser is one in which any substring in the sentential form that matches a right-hand side in the grammar is a handle. Determine conditions for which this parser works. See also Problem 9.1.

Problem 10.10: On page 361 we wrote that $\text{FOLLOW}_{\text{LM}}$ is in general neither a subset nor a superset of FOLLOW; explain.

Problem 10.11: *Project:* As we saw on page 364, the NSLR table generation process can produce unreachable states. Design an algorithm to find and remove them, or an algorithm that does not create them in the first place.

Problem 10.12: Since the $\text{FOLLOW}_{\text{LM}}$ set can contain symbols that cannot actually follow a given item in a given set, it is possible that the NSLR(1) parser introduces an item for a look-ahead non-terminal, which, when it is finally recognized in an incorrect input string, does not allow being shifted over, giving rise to an error message of the type "Syntactic entity *X* cannot appear here". Construct a grammar and input string that shows this behavior.

Problem 10.13: When we abandon an active item in an LR(k,∞) parser and find that there is no dotted look-ahead to be promoted to active item, all is not lost. We could: 1. scan the look-ahead to find the first non-terminal, which will be at position $k_1 > k$, and try parsing again, this time as LR(k_1,∞); 2. hope that the rest of the input matches one of the other items in the state, so we can at least parse *this* input. Develop and evaluate both ideas.

Problem 10.14: Suppose we reach the end of the input in an LR(k,∞) parser and we still have more than one item left. Design an algorithm that constructs the multiple parse trees from the state of the parser at that moment.

Problem 10.15: *Project:* Design and implement a complete, linear-time, LR(k,∞) parser.

Problem 10.16: Complete the Partitioned LR automaton of Figure 10.15.

Problem 10.17: Suppose there is a shift/reduce conflict in a Partitioned LR parser. Design a way to adapt the grammar to Partitioned LR parsing.

Problem 10.18: *Project:* As with the NSLR table generation process, the Partitioned LR table generation process as described in Section 10.2.4.2 can produce unreachable states. Design an algorithm to find and remove them, or an algorithm that does not create them in the first place.

Problem 10.19: *Project:* Design and implement a complete Partitioned LR parser.

Problem 10.20: *Project:* Since head-corner parsers require the nodes of the parse tree to be constructed in non-canonical order, it seems an ideal candidate for non-canonical parsing. Design a non-canonical LR parsing algorithm that postpones the reduction until it can reduce the head of a rule.

11

Generalized Deterministic Parsers

Generalized deterministic parsers are general breadth-first context-free parsers that gain efficiency by exploiting the methods and tables used by deterministic parsers, even if these tables have conflicts (inadequate states) in them. Viewed alternatively, generalized deterministic parsers are deterministic parsers extended with a breadth-first search mechanism so they will be able to operate with tables with some multiple, conflicting entries. The latter view is usually more to the point.

Before going into the algorithms, we have to spend a few words on the question what exactly constitutes a "generalized deterministic parser". Usually the term is taken to mean "a parser obtained by strengthening an almost-deterministic parser by doing limited breadth-first search", and initially the technique was applied only to parsers with LR tables with just a few inadequate states. Later research has shown that "generalized parsing" can also be used profitably with tables with large amounts of inadequate states, and even without tables (actually with trivial tables; see next paragraph). We will therefore cover under this heading any breadth-first CF parser with some, even the weakest, table support.

Trivial parse tables are interesting in themselves, since they are a low extreme all other tables can be measured against: they form the bottom element if one wishes to order parse tables in a lattice. The *trivial bottom-up table* has one state, which says: both shift the next input token onto the top of the stack and reduce with all rules whose right-hand sides match the top of the stack. The *trivial top-down table* has one state, which says: either match the next input token to the top of the stack or predict with all rules whose left-hand sides match the top of the stack. Since states are used to make a distinction and since just one state cannot make a distinction, one can also leave it out.

As said above, the first generalized parsing algorithms that were designed were based on almost-LR parse tables, and much more is known about generalized LR parsing than about the other variants. We shall therefore treat generalized LR parsing first, and then those based on other tables.

Merrill [171] has shown that it is also possible to do generalized LR parsing by strengthening an almost-deterministic LR parser by doing *depth-first* search.

11.1 Generalized LR Parsing

Section 9.4 has shown how we can make very powerful and efficient parsers for LR grammars. But although most languages of practical interest are LR (deterministic) most grammars of practical interest are not. And if we try to design an LR grammar for one of these LR languages we find that such a grammar is hard to construct, or does not provide the proper structuring needed for the semantics, or — usually — both. This limits the practical usefulness of LR parsing.

On the bright side, most practically useful grammars are *almost* LR, which means that their LR automata have only a few inadequate states. So we become interested in ways to exploit such almost-LR automata, and the answer lies in reintroducing a little bit of breadth-first search (Section 7.1.2). Research in this direction has resulted in "generalized LR parsing".

Generalized LR parsing (*GLR parsing*) can be characterized as left-to-right bottom-up breadth-first parsing in which the breadth-first search is limited by information from an LR handle-recognizing automaton; the LR automaton is allowed to have inadequate states (conflicts) in it.

GLR parsing was first described by Lang [159] in 1974 but unfortunately the publication was not noticed by the world. The idea was rediscovered in 1984 by Tomita [160, 161], who wrote a 200-page book about it [162]. This time the world took notice and the technique became known as *Tomita parsing*. Over the years it was increasingly found that this naming was not ideal, and the technique is now almost universally referred to as "GLR parsing".

11.1.1 The Basic GLR Parsing Algorithm

The GLR method does breadth-first search exactly over those parsing decisions that are not solved by the LR automaton (which can be LR(1), LALR(1), SLR(1), LR(0), precedence or even simpler), while at the same time keeping the partial parse trees in a form akin to the common representation of Section 7.1.3. More precisely, whenever an inadequate state is encountered on the top of the stack, the following steps are taken:

1. For each possible reduce in the state, a copy of the stack is made and the reduce is applied to it. This removes part of the right end of the stack and replaces it with a non-terminal; using this non-terminal as a move in the automaton, we find a new state to put on the top of the stack. If this state again allows reductions, this copy step is repeated until all reduces have been treated, resulting in equally many stack copies.
2. Stacks that have a rightmost state that does not allow a shift on the next input token are discarded (since they resulted from incorrect guesses). Copies of the next input token are shifted onto the remaining stacks.

There are a number of things to be noted here. First, if the automaton uses lookahead, this is of course taken into account in deciding which reduces are possible in step 1; ignoring this information would not be incorrect but would cause more

stacks to be copied and subsequently discarded. Second, the process in step 1 may not terminate. If a grammar contains non-terminals that can produce themselves, for example $A \xrightarrow{*} A$ (loops), A will continuously be reduced to A. And grammars with hidden left recursion turn out to cause infinite numbers of ε-reductions. These two problems will be dealt with in Section 11.1.3. Third, if all stacks are discarded in step 2 the input was in error, at that specific point. Fourth, if the table is weak, especially if it is not produced by an LR process, it may suggest reductions with rules whose right-hand sides are not present on the stack; such reductions are then to be ignored.

11.1.2 Necessary Optimizations

The above steps form the basic mechanism of the GLR parser. Since simple stack duplication may cause a proliferation of stacks and is apt to duplicate much more data than necessary, two optimizations are used in the practical form of the parser: combining equal stack suffixes and combining equal stack prefixes. We shall demonstrate all three techniques using the grammar of Figure 11.1 as an example. The grammar

```
S_S  →  E $
 E   →  E + E
 E   →  d
```

Fig. 11.1. A moderately ambiguous grammar

is a variant of that of Figure 3.1 and is moderately ambiguous. Its LR(0) automa-

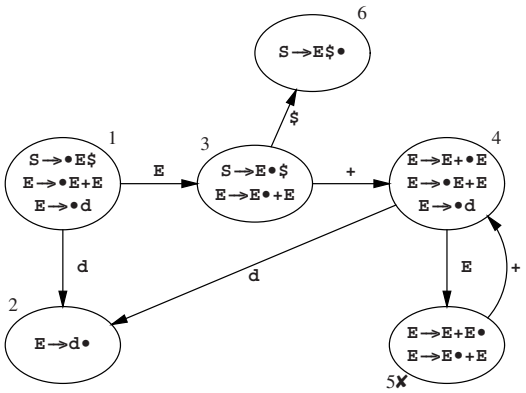


Fig. 11.2. LR(0) automaton to the grammar of Figure 11.1

ton is shown in Figure 11.2; it has one inadequate state, ⑤. Since the grammar is ambiguous, there is no point in using a stronger LR method. For more (and larger!) examples see Tomita [162] and several of the publications in (Web)Section 18.2.1.

11.1.2.1 Stack Duplication

Refer to Figure 11.3, in which we assume the input **d+d+d$**. The automaton starts

a	①															**d+d+d$**	shift
b	①	**d**	②													**+d+d$**	reduce, shift, shift
c	①	**E**	③	**+**	④	**d**	②									**+d$**	reduce
d	①	**E**	③	**+**	④	**E**	⑤									**+d$**	duplicate to *e1* and *e2*; reduce *e1*
e1	①			**E**			③									**+d$**	shift, shift, to *f1*
e2	①	**E**	③	**+**	④	**E**	⑤									**+d$**	shift, shift, to *f2*
f1	①			**E**			③	**+**	④	**d**	②					**$**	reduce, to *g1*
f2	①	**E**	③	**+**	④	**E**	⑤	**+**	④	**d**	②					**$**	reduce, to *g2*
g1	①			**E**			③	**+**	④	**E**	⑤					**$**	duplicate to *h1.1* and *h1.2*; reduce *h1.1*
g2	①	**E**	③	**+**	④	**E**	⑤	**+**	④	**E**	⑤					**$**	duplicate to *h2.1* and *h2.2*; reduce *h2.1*
h1.1	①					**E**					③					**$**	shift, to *i1.1*
h1.2	①			**E**			③	**+**	④	**E**	⑤					**$**	discard
h2.1	①	**E**	③	**+**	④			**E**			⑤					**$**	reduce again, to *h2.1a*
h2.2	①	**E**	③	**+**	④	**E**	⑤	**+**	④	**E**	⑤					**$**	discard
h2.1a	①			**E**							③					**$**	shift, to *i2.1a*
i1.1	①					**E**					③	**$**	⑥				reduce, to *j1.1*
i2.1a	①			**E**							③	**$**	⑥				reduce, to *j2.1a*
j1.1	①							**S**									accept
j2.1a	①			**S**													accept

Fig. 11.3. Sequence of stack configurations while parsing **d+d+d$**

in state ① (frame *a*). The steps shift (*b*), reduce, shift, shift (*c*) and reduce (*d*) are problem-free and bring us to state ⑤. The last state, however, is inadequate, allowing a reduce and a shift. True to the breadth-first search method and in accordance with step 1 above, the stack is now duplicated and the top of one of the copies is reduced (*e1*) while the other one is left available for a subsequent shift (*e2*). Note that no further reduction is possible and that both stacks now have a different top state. Both states allow a shift and then another (*f1*, *f2*) and then a reduce (*g1*, *g2*). Now both stacks carry an inadequate state on top and need to be duplicated, after which operation one of the copies undergoes a reduction (*h1.1*, *h1.2*, *h2.1*, *h2.2*). It now turns out that the stack in *h2.1* again features an inadequate state ⑤ after the reduction; it will again have to be duplicated and have one copy reduced. This gives the stack in *h2.1a*. Now all possible reductions have been done and it is time for a shift again. Only state ③ allows a shift on **$**, so the other stacks are discarded and we are left with *i1.1* and *i2.1a*. Both require a reduction, yielding *j1.1* and *j2.1a*, which are accepting states. The parser stops and has found two parsings.

In order to save space and to avoid cluttering up the pictures, we have not shown the partial parse trees that resulted from the various reductions that have taken place.

If we had done so, we would have found the two **S**s in *j.1.1* and *j.2.1a* holding the parse trees of Figure 11.4.

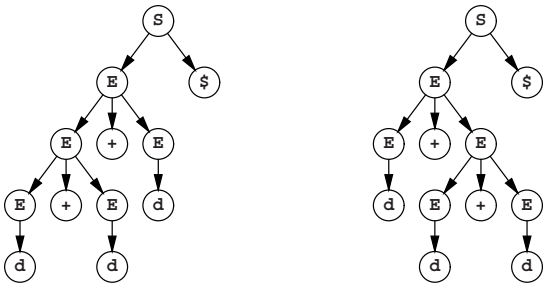


Fig. 11.4. Parse trees in the accepting states of Figure 11.3

11.1.2.2 Combining Equal Stack Suffixes

Examining Figure 11.3 *f1* and *f2*, we see that once both stacks have the same state on top, further actions on both stacks will be identical, and the idea suggests itself to combine the two stacks to avoid duplicate work. This approach is depicted in Figure 11.5(*f*) (Figure 11.5(*a*) to (*e*) are identical to those of Figure 11.3 and are not shown). Note that we have not only combined the two ②s on top, but also the two ④s one token back; this is possible because they are both connected by the same token, **d**. In fact, we are combining equal stack suffixes.

Combining equal stack suffixes is, however, not entirely without problems, as becomes evident as soon as we need to do a reduce that spans the merge point. This happens in (*g*), which also features an inadequate state. Now a number of things happen. First, since the state is inadequate, the whole set of combined stacks connected to it are duplicated. One copy (*g"*) is intended for the shift in step 2, but there is no shift from ⑤ over **$**, so the whole copy is discarded. The other (*g'*) is subjected to the reduce. This reduce, however, spans the merge point (state ④) and extends up both stacks, comprising a different leftmost **E** in the two branches. To perform it properly, the stack combination that gave rise to (*f*) is undone, resulting in (*g'.1*) and (*g'.2*). The reduces are then applied to both stacks, resulting in (*h1*) and (*h2*). The reduce in (*h2*) again puts the inadequate state ⑤ on top, which necessitates another copy operation, to (*h2.1*) for the reduce, and to (*h2.2*) for the shift. The reduce on (*h2.1*) results in the stack ①**E**③, which has a ③ on top. But we already have a stack with a ③ on top: (*h1*), so we must combine their tops. We then see that the combined top is connected by two **E**s to two ①s, so we can combine these two, which combines the whole stacks: the result of the reduction of (*h2.1*) is (*h1*). This shifts to (*i*) and from there the road is clear. This is the parser described in Tomita's first publication [160].

Although in this example the stack combinations are undone almost as fast as they are performed, stack combination greatly contributes to the parser's efficiency in the general case. It is essential in preventing exponential growth.

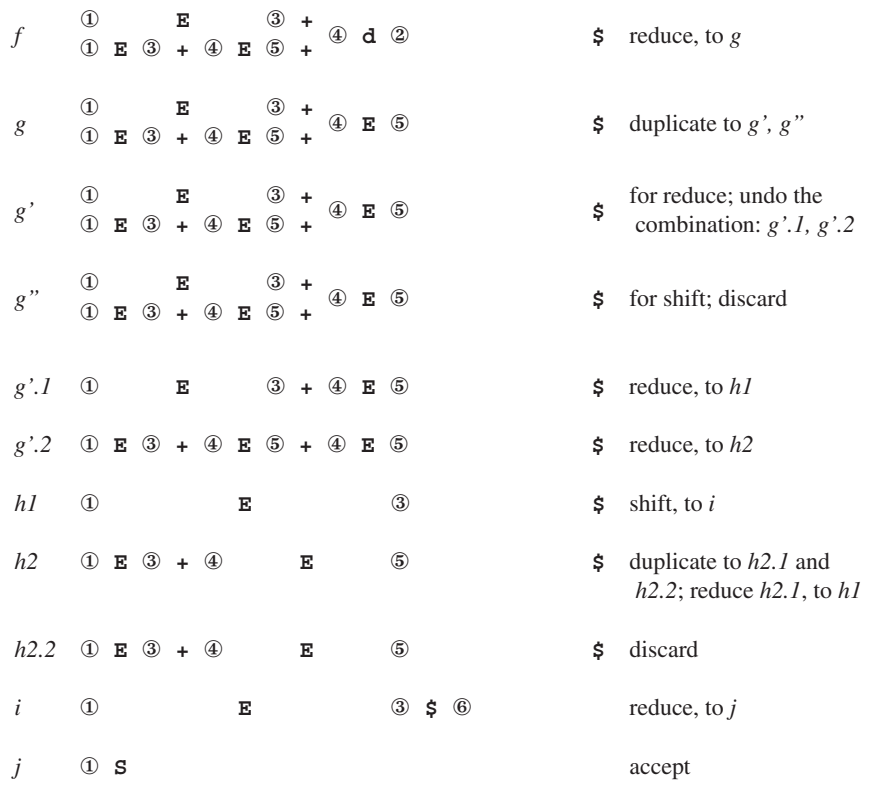

f
① E ③ + ④ E ⑤ + ④ d ② $ reduce, to g
g
① E ③ + ④ E ⑤ + ④ E ⑤ $ duplicate to g', g"
g'
① E ③ + ④ E ⑤ + ④ E ⑤ $ for reduce; undo the combination: g'.1, g'.2
g"
① E ③ + ④ E ⑤ + ④ E ⑤ $ for shift; discard
g'.1
① E ③ + ④ E ⑤ $ reduce, to h1
g'.2
① E ③ + ④ E ⑤ + ④ E ⑤ $ reduce, to h2
h1
① E ③ $ shift, to i
h2
① E ③ + ④ E ⑤ $ duplicate to h2.1 and h2.2; reduce h2.1, to h1
h2.2
① E ③ + ④ E ⑤ $ discard
i
① E ③ $ ⑥ reduce, to j
j
① S accept

Fig. 11.5. Stack configurations with equal-suffix combination

In the process of combining stack suffixes we have lost the power to construct the parse trees on the fly: we have combined **E**s that represent different parse trees. This might actually be a good thing. After all, in GLR parsing many stacks will be discarded, and the work spent in constructing the parse trees in them is wasted. And the parse trees can be retrieved afterwards, in the same way as explained for the Earley parser (Section 7.2.1.2). More semantic information may be available then, which will reduce the number of parse trees. It is of course also possible to continue assembling parse tree information, but then fewer opportunities to combine equal suffixes will be available.

11.1.2.3 Combining Equal Stack Prefixes

When step 1 above calls for the stack to be copied, there is actually no need to copy the entire stack; just copying the top states suffices. When we duplicate the stack of Figure 11.3(*d*), we have one forked stack for (*e1*) and (*e2*):

Now the reduce is applied to one top state ⑤ and only so much of the stack is copied as is subject to the reduce:

	E ③	shift
e' ①		**+d$**
	E ③ **+** ④ **E** ⑤	shift

In our example almost the entire stack gets copied, but if the stack is somewhat larger, considerable savings can result. This is the parser described in Tomita's second publication [161].

The title of this section is in fact incorrect: in practice no equal stack prefixes are combined, they are never created in the first place. The pseudo-need for combination arises from our wish to explain first the simpler but impractical form of the algorithm in Section 11.1.1. A better name for the technique would be "common stack prefix preservation".

11.1.2.4 The Graph-Structured Stack

The stack in the GLR algorithm and its optimizations has the form of a "dag", a directed acyclic graph. It is called a *graph-structured stack*, often abbreviated to *GSS*. Since stacks are processed from the top downwards in most algorithms, the edges of nodes in them point to their predecessors, although a bidirectional graph representation is occasionally useful. The GSS corresponding to the sample parsing in the previous sections is shown in Figure 11.6.

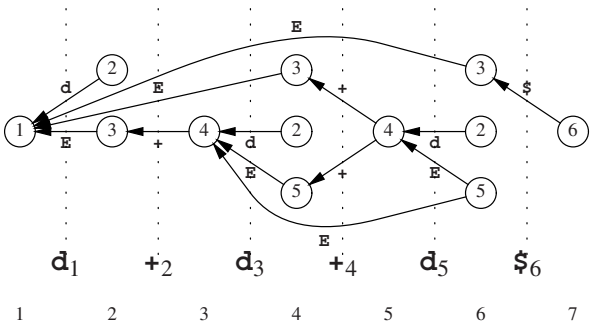


Fig. 11.6. Graph-structured stack for the parsing of **d+d+d$**

11.1.3 Hidden Left Recursion and Loops

Two grammar features threaten the correctness of the GLR parser: hidden left recursion and loops. Both features can cause infinite reduce loops and thus cause step 1 of the GLR algorithm not to terminate. That grammar loops cause reduce loops is easy to see: with rules $A \rightarrow B$ and $B \rightarrow A$ and having found a B, we can reduce the B to A

using the first rule and then reduce the A to B by the second, getting our original B back. Note that the stack does not grow in this process.

The problem with hidden left recursion is less obvious, all the more since visible left recursion is OK. Suppose the grammar has a rule $A \rightarrow \alpha A \beta$, where α produces ε. Then some state s contains the item $A \rightarrow \alpha \bullet A \beta$ (item i_1), and so it also contains $A \rightarrow \bullet \alpha A \beta$ (item i_2). When we start parsing in state s. an α will be recognized, which moves the dot over it in item i_2, which makes it equal to item i_1, and we are back where we were. If α was recognized as matching ε, we are still in the same place in the input and the parser is again poised to recognize an α. Nothing has changed except that the stack now shows a recognized α. So a loop results. We note two things: with hidden left recursion the stack grows; and visible left recursion is no problem because no reduction is involved: item i_2 is the same as item i_1 in state s.

By a stroke of good luck the optimizations from Sections 11.1.2.2 and 11.1.2.3 help a great deal in solving both problems. To show how this works we use the grammar

$$\begin{array}{lcl} \mathbf{S}_S & \rightarrow & \mathbf{A\ S\ c} \\ \mathbf{S} & \rightarrow & \mathbf{b} \\ \mathbf{A} & \rightarrow & \varepsilon \end{array}$$

It produces the language $\varepsilon^n \mathbf{bc}^n$. This is of course $\mathbf{bc}^n$, since the leading εs are invisible, which is exactly the point. Its LR(0) automaton is in Figure 11.7; states ① and ② contain conflicts. We now see clearly the root of the trouble: the transition

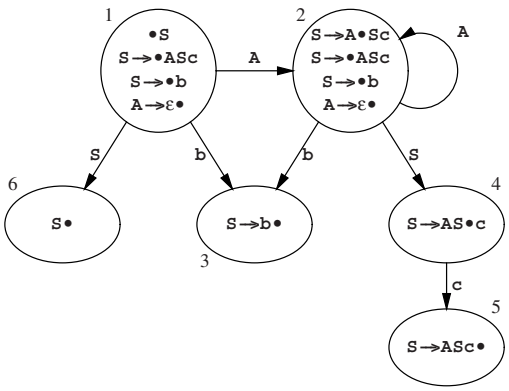


Fig. 11.7. LR(0) automaton of a grammar with hidden left recursion

diagram has a cycle the edges of which consist only of nullable non-terminals (**A**s in our case), and traveling those edges can bring us back to the same state without consuming input. Or, in other words, the automaton cannot know how many **A**s to stack to match the coming sequence of **c**s.

We use the GLR parser based on this automaton to parse the input **bccc**; the resulting GSS is in Figure 11.8. Most of the interesting things happen already before the **b** is shifted, in position 1; the four columns 2…5 only serve to show that any

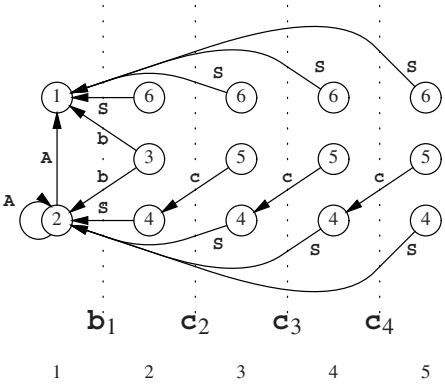


Fig. 11.8. Graph-structured stack for parsing with a grammar with hidden left recursion

number of **c**s can be accepted. The GLR parser starts in state ①, which calls for a reduction, **A**→ε, leading to a ② in the same position. State ② also calls for a reduction, **A**→ε, again leading to a ②. But we already have a ② in this position, so we combine them. Note that this creates a cycle in the GSS, a feature it did not possess before; in fact, the stack can be represented by a regular expression: ①**A**(②**A**)*. There is another stack segment starting with **A** from ② but it ends in ① so we cannot combine it with our new stack segment.

The next round of step 1 of the GLR algorithm yields another stack segment from ② over **A** to ②, so now it combines with the previous one, and no modification to the GSS is made. This means that if we go on doing reduces, nothing will change any more and step 1 will just loop. So we modify step 1 by replacing the condition "until all reduces have been treated" by "until no reduce modifies the GSS any more". This introduces a transitive closure component.

Step 2 shifts over the **b** to position 2 and introduces the stack segments ①**b**③ and ②**b**③; our stack is now represented by ①**b**③ | ①**A**(②**A**)*③. State ③ is a reduce state, yielding an **S**, which gets shifted. The one starting at ① leads to ⑥; the one starting at ② to ④. A shift over the first **c** gives a ⑤ in position 3, which is again a reduce state which wants to reduce **A**S**c** to **A**. Working back from the ⑤, we find two paths **A**S**c** back, one leading to ① and the other to ②, the latter going through the **A** loop once. Of course both states allow a shift on the **S** from the reduction, the first yielding a ⑥ and the second a ④, both in position 3. The ④ allows the next **c**, $\mathbf{c}_3$ to be shifted, and the resulting ⑤ causes the two-path reduction to be done again. After shifting the last **c** and doing the reductions, an accepting state ⑥ remains, so the input is recognized. We see that the (②**A**)* in position 1 can produce any desired number of **A**s needed by **c** further on in the input. This is an application of the left context (Section 9.12.1).

Nozohoor-Farshi [167] notes that the cyclic structures resulting from ε-reductions in hidden left recursion can be precomputed and need not be computed repeatedly at parse time.

A different way of handling hidden left recursion and loops is to employ the ε-LR automaton of Nederhof and Sarbo [94] (Section 9.5.4) as the underlying automaton. Since each stack entry in this parser corresponds to a non-empty segment of the input, a reduce loop in step 1 of the GLR algorithm cannot occur. As said in that section, it complicates the reduce action itself, though.

11.1.4 Extensions and Improvements

The basic GLR parser has exponential time requirements, but the standard GLR parser, which includes equal stack prefix and suffix combination, is $O(n^{k+1})$, where k is the length of the longest right-hand side in the grammar. To limit this to $O(n^3)$, Kipps [165] stores distance information in the GSS so all the starting points of a reduce of length k can be found quickly. The same effect is achieved by Scott et al. [182] by using an automaton that does reductions of length 2 only. Remarkably, Alonso Pardo et al. [175] derive an $O(n^3)$ GLR parser from an Earley parser.

Most other speed improvements are based on improving the underlying LR parser. Nullable non-terminals at the beginning of a rule cause problems when they hide left recursion; those at the end cause inefficiencies because they postpone simplifying reductions. Algorithms to incorporate both kinds of ε-moves in the automaton are reported by Scott et al. [177], Aycock et al. [178], Johnstone and Scott [180]. The resulting parser, called a *RNGLR* parser, for "Right-Nullable GLR", is very efficient on a wide variety of grammar; for an extensive description see Scott and Johnstone [188].

The number of stack operations — the most expensive operations in a GLR parser — can be reduced considerably by using the stack for recursion only; all other stack operations can be incorporated in the LR automaton, as described in Section 9.10.2. Algorithms to this effect are discussed by Aycock and Horspool [176], Johnstone and Scott [181] and Scott and Johnstone [183, 186].

The applicability of this optimization is hindered by the very large size of the tables that result for grammars for every-day programming languages like C++, but it turns out that table size can sometimes be reduced spectacularly by allowing a little bit more stack activity than needed. Trade-offs are can mapped by Johnstone and Scott [185].

Johnstone et al. [187] gives an overview of the various speed-up techniques.

There has been some speculation about what strength of LR table to use: LR(0), SLR(1), LALR(1) or LR(1); most authors and practitioners settle for LR(0), because it is the easiest table to construct. In a low-availability paper, Lankhorst [166] reports experiments which show that LALR(1) is best, but LR(0) is only 5-10% worse. LR(1) especially does much worse; its large number of states causes the GSS to be very frayed, causing much overhead. More specialized LR automata are considered by Johnstone et al. [184], with RNGLR a likely winner.

An incremental GLR parser is described by Vilares Ferro and Dion [331]. In [179] Fortes Gálvez et al. show the construction of a generalized DR parser (Section 9.13.1). It required rethinking the DR parsing process.

Piastra and Bolognesi [168] outline a way to efficiently perform semantic actions during GLR parsing.

11.2 Generalized LL Parsing

For a long time it was thought that generalized LL parsing could not exist: an LL parser in which the conflicts were caused by left recursion could never work since it would always loop, regardless of how much breadth-first search one would do. However, in 1993 van Deudekom and Kooiman [170] reported a top-down generalized LL parser, in a report on the implementation of non-correcting error recovery according to Richter [313].

The algorithm we describe here is a simplification of theirs, the reason being that their parser is integrated with the creation of a suffix grammar and is adapted to the idiosyncrasies of *LLgen*, the LL(1) parser generator in which it is embedded. Those aspects are discussed in annotation [320].

11.2.1 Simple Generalized LL Parsing

The generalized LL parser is very loosely based on Greibach's 1964 "directed production analyser" [7], which is actually a breadth-first predictive parser:

- As long as the top of the prediction stack is a non-terminal, we predict a right-hand side for it, based on the LL parse table; this is the prediction phase. Then we match the input symbol to the top of the stack; this is the matching phase.
- If the LL(1) table presents more than one right-hand side for a non-terminal *A*, the prediction stack forks, resulting in more than one top and making the node for *A* the parent of more than one substack. We apply the above to all tops.
- This leads to a prediction tree rather than a prediction stack; technically it is a reversed tree, "reversed" because the parent pointers point from the leaves to the root. The leaves together are kept in a "top list".
- If we are about to make a prediction for a non-terminal *A* which has already been expanded before in the same branch of the prediction tree in the same prediction session, we replace the prediction by back pointer, forming a loop in the tree.
- This leads to a prediction tree with loops; note that this is still less general than an arbitrary graph.

Greibach's parser as described in [7] works only for grammars in Greibach Normal Form (Section 6.2), thus avoiding the use of an LL(1) table and problems with left recursion. The admission of left recursion and its treatment as suggested above is relatively simple, but it turns out that its combination with ε-rules causes a host of problems.

In the absence of left recursion and ε-rules, things are straightforward. Given the grammar

```
S_S  ->  AB
 A   ->  a | X
 B   ->  b
 X   ->  a
```

and the input **ab**, we get the predictions shown in Figure 11.9. The LL(1) table (not

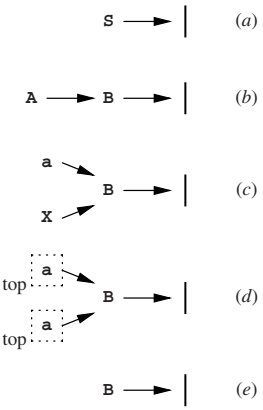


Fig. 11.9. Top-down predictions as reversed trees

given here) predicts **S→AB** for (**S**,**a**) (Figure 11.9(*b*)), and both **A→a** and **A→X** for (**A**,**a**) (*c*); next **X** is expanded (*d*, where the top list is marked by the dotted box). The input **a** matches both tops, after which only the prediction **B** remains (*e*); it predicts **B→b**, after which the input **b** is matched. If we keep the grammar rules used, we obtain two parse trees.

There are two ways to perform the predictions: depth-first and breadth-first. Suppose we are about to predict expansions for a non-terminal A, and the LL(1) table indicates the alternatives $A_1 \ldots A_n$. In depth-first prediction we first stack A_1 and if its top is another non-terminal B, we predict and expand B recursively, until all new branches have terminal symbols on top, which we then enter into the top list, or until we are stopped by left recursion. We then expand A_2 in the same way as the second prong of the fork, etc. In breadth-first prediction we stack the alternative $A_1 \ldots A_n$, one next to another in fork fashion, and append the top of each stack to the top list. Each of these tops will then get the rest of their treatment when their turn comes in the processing of the top list.

It does not make much difference which method we use, since eventually the same actions are performed, and a very similar data structure results. The action are, however, performed in a different order, and it is good to keep that in mind for the rest of the algorithm. Van Deudekom and Kooiman use depth-first prediction; breadth-first prediction makes its easier to handle infinite ambiguity (Section 11.2.3).

11.2.2 Generalized LL Parsing with Left-Recursion

When we introduce left recursion this simple data structure begins to shows its shortcomings. Given the grammar

```
S_S  →  AB
 A   →  a | S
 B   →  b
```

and the input **abb**, we get the predictions **aB** and **SB**, sharing the **B** (Figure 11.10(*a*)). But when we want to expand the left-recursive **S** by making a loop in the prediction tree as suggested above, two questions arise: how do we know that this is the left-recursive expansion and not the initial expansion, and where is the node to which we should connect the loop? Both questions are answered by the same device: we keep a non-terminal node after its expansion, properly marked as "expanded", as shown in Figure 11.10(*b*) where square brackets around a non-terminal indicate that it has

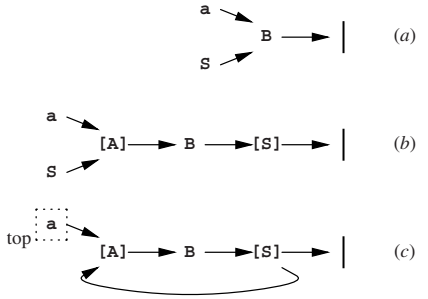


Fig. 11.10. Prediction trees without expanded nodes (*a*), with expanded nodes (*b*), and with a loop (*c*)

already been expanded.

We have to modify our parsing algorithm to accommodate the expanded nodes, as follows. When such an expanded node for a non-terminal *A* appears in the top list during parsing, it means that a terminal production of *A* has been recognized. The expanded node is skipped, and the node or nodes pointed to by its outgoing pointers are included in the top list in its place.

With this data structure we can check the top **S** for being a left-recursive occurrence by following the parent pointers down the stack to see if **S** has been expanded before in this prediction phase; and the expanded node can act as the starting point of the loop (*c*). The test requires us to know if a non-terminal was expanded during the present prediction phase; we can achieve this, for example, by keeping a set of all non-terminals expanded in the present prediction phase, or by numbering the prediction phases increasingly and recording the prediction phase number in each expanded node.

Note the direction of the back pointer: like all other pointers in the reversed tree it runs from the predicted node "back" to the predicting node. Note also that parent

pointers and back pointers are treated differently: during parsing we follow both, but during left-recursion checking we only follow the parent pointer.

It is clear that the nodes in the loop in Figure 11.10(*c*) may be used more than once. This requires us to be more careful with them during prediction: rather than just marking the node "expanded", we need to make a copy of it, and mark the copy; we then base our prediction on that copy.

We can see this mechanism in action when we continue the parsing of **abb**. We start from Figure 11.11(*a*), which is a copy of Figure 11.10(*c*) except that the original

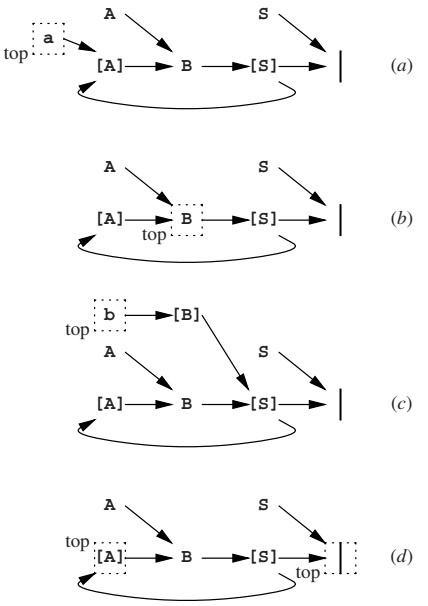


Fig. 11.11. Prediction trees with copied expanded nodes

nodes for **S** and **A** are still visible. Note that these are not in the top list; they play no direct role in the algorithm any more, but it is good to remember they are there. The first matching phase matches the **a**, which makes the expanded node **[A]** the only member of the next top list. The **[A]** node has already been expanded (an **A→a** was recognized), we follow its pointer, and bring node **B** into the top list (*b*). Non-terminal **B** and look-ahead **b** command the prediction **B→b**, so a marked copy is made of the node and the prediction **b** is based on it and its first component, the **b**, is now in the top list (*c*).

The **b** is matched, the expanded **[B]** is skipped, and we arrive at the node **[S]**. This node, unlike the nodes we have seen up to now, has more than one outgoing pointer. The parent pointer points to the bottom of the stack, indicating that the input could be finished here (indeed **ab** is produced by the grammar), and the back pointer leads to **[A]**. We follow both, so the bottom of the stack and the **[A]** node together

form the next top list (*d*). The LL(1) parsing table immediately rejects the bottom of the stack, and the `[A]` is skipped because it has already been expanded (a `A`→`S` has been recognized). Now node `B` is the only member of the new top list; this brings us back to Figure 11.11(*b*), and the system is ready to match the next `b`.

This technique can handle the ultimate in left recursion, a loop in the grammar, but only with some difficulty. Prediction for the grammar `S`→`S`, `S`→`a` and input `a` ties the parser in an infinite loop if we do not take measures. The initial step predicts

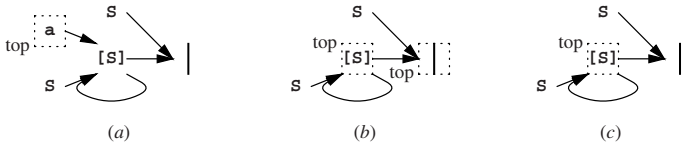


Fig. 11.12. Predictions with a non-ε grammar loop

both rules for `S`; the first one is left-recursive and causes a loop in the prediction tree. The result is shown in Figure 11.12(*a*). Matching the `a` causes the `[S]` node to become the only member of the top list; its processing follows parent and back pointers, and appends the bottom of the stack and the node `[S]` itself again to the top list (*b*). The node `[S]` is then processed again, etc., resulting in a parser loop, as shown in Figure 11.12(*c*).

This is not actually a fault of the algorithm. The algorithm tries faithfully to find all possible parsings, but the grammar is infinitely ambiguous, and the algorithm is just building the data structure for a infinite number of parsings. The problem can be solved crudely by not appending a node to the top list if it is already there. This does, however, ruin the property that the parser finds all possible parsings; the next section gives a much sharper criterion for the suppression of top nodes, which also solves the grammar loop problem.

11.2.3 Generalized LL Parsing with ε-Rules

The introduction of ε-rules to the parser described above causes hardly any problems as long as no left-recursion is involved. When the LL(1) parse table indicates an ε-rule $A \rightarrow \varepsilon$ as the prediction for an A on the top of the stack, its empty right-hand side is stacked on the expanded $[A]$ node, which causes this node to be appended to the top list. When its turn comes, it is skipped and its parent and back pointer nodes are appended to the top list. This is exactly what should happen with a nullable non-terminal.

The situation is more complicated for a left-recursive nullable non-terminal. Suppose A is such a non-terminal. As above, a node $[A]$ is appended to the top list when the system discovers that the node for A produces ε. Later it is then skipped and its parent and back pointer nodes are appended to the top list; but the problem is that its list of back pointers may be incomplete. After all that list is still under construc-

tion, and it may happen that there is a left-recursion loop that is discovered after the nullable $[A]$ node has been processed. This can make us miss parsings.

There are several ways to solve this problem. Van Deudekom and Kooiman derive nullability information from the LL(1) parser generator, and if a left-recursion back pointer is constructed for a nullable node $[A]$ pointing to a node B, that node is appended to the top list right away. This makes sure that no back pointers are missed. Also, when an expanded nullable node created in the present prediction phase comes up for prediction its back pointers are *not* followed to avoid spurious parsings. (Note that if such a node has back pointers it must be left-recursive.) For a second way of solving this see Problem 11.3.

A much more severe problem arises when there is a loop in the grammar involving a nullable non-terminal. Complicated cases like

```
S_S  →  S T U
 T   →  T U S
 U   →  U S T
 S   →  ε
 T   →  ε
 U   →  ε
 S   →  a
```

form convoluted, ever-growing prediction graphs, full of predicted productions of length 0. Van Deudekom and Kooiman [170] do not address this problem; we shall sketch a solution here, under the assumption that prediction is performed breadth-first.

The grammar is infinitely ambiguous in many ways, which causes the algorithm to produce infinitely many parsings. These infinitely many parsings originate from the second, third, etc., appearance of a node in the top list. As said before, bluntly refusing to take more than one copy of a node in the top list damages the ability to produce the correct number of parsings in non-pathological cases. So we need a better criterion for deciding to admit an expanded node N to the top list.

To this end we follow the parent and back pointers from N and put them in a set ε_N, the set of nodes reachable from N by ε-transitions. So far it is the set of nodes that will be added to the top list when node N will be processed. Now for each nullable or expanded node M in the set we put the parent and back pointers of M in ε_N and we continue doing so until no more nodes are to be added (this is another example of a transitive closure algorithm).

The set ε_N thus obtained is the set of nodes that would be predicted by N without predicting an intervening token, if N is appended to the top list. Nodes that cause a prediction which includes at least one token or non-nullable non-terminal cannot give rise to prediction loops, since they cannot lead to the recognition of a non-terminal expanded in the present prediction phase; so the set ε_N contains the nodes that are not safe.

Two tests can be made on this set. If the original node N is not in ε_N, node N can safely be appended to the top list; there is no prediction loop involving N and thus no parser loop needs to be feared from N. If node N *is* in ε_N, there is a prediction loop from N to itself; appending N is unsafe, but not appending it may make us lose

parsings. With this information the decision is easy: if ε_N contains a node M for the same non-terminal as node N, and M is already in the top list, then we can afford not to append node N to the top list. The reason is that the predictions for M are the same as for N, and appending N would just produce them a second time. If there is no such node M, N must be appended to the top list; next time it will produce a similar ε_N set with different nodes, but now N is in the top list and N will not be appended a second time, thus quenching the loop.

A number of remarks can be made about this algorithm. The first is that it is immaterial whether the node M is in the top list before or after the node being processed. If it is before, it has already been expanded; if it is after, it will be expanded; in both cases parsings due to M will not be lost. The second is that the algorithm works with breadth-first prediction only. The top list in depth-first prediction contains terminals and symbols revealed by matching rather than the in-between results of predictions, and an additional data structure is needed to make the algorithm work with depth-first prediction.

The third is that we do not need to actually construct the set ε_N. While scanning the prediction graph by following nullable and expanded nodes, we can record the answers to our two questions: do we meet node N, and do we find a node M with the properties described above. Unless both answers are affirmative, we append the node for N to the top list.

And last but not least, it is a bit worrying that the algorithm was designed by repairing the shortcomings of a simple approach; also, no correctness proof of the algorithm has been published to our knowledge. It is probably fair to say that this is the least researched and least well-founded algorithm in the book.

Van Deudekom and Kooiman [170] describe various optimizations in their report, including combining identical nodes in the top list. This turns their data structure into what could be called "a directed acyclic graph with restricted cycles" and they describe a specialized garbage collector for it.

11.2.4 Generalized Cancellation and LC Parsing

Similar data structures are used by Nederhof to implement generalized cancellation parsing [105] and left-corner parsing [172].

Cancellation parsing (Section 6.8) can handle direct left recursion, but it cannot handle indirect (hidden) left recursion, so loops in the prediction tree will still occur.

In addition to allowing all CF grammars, generalized cancellation parsing has the advantage over deterministic cancellation parsing that it can work with a simplified parse table that derives directly from the grammar. This allows a very simple and light-weight implementation in Prolog, for which see [105].

The LC version uses the left-corner relation $\angle$ (Section 10.1.1.5) rather than the actual LC parse table. It lends itself well for serious optimization, including such things as precomputation of ε-generating parse forests. It is all described in [172].

11.3 Conclusion

The generalized non-deterministic parsing algorithms extend the powerful deterministic LR and LL techniques to general CF grammars. For grammars with limited non-determinism, which includes most practical grammars, they are very efficient and vastly outperform CYK and Earley parsers, usually achieving almost linear time dependency.

Their basic tool is the forked stack, a forked reduction stack for GLR and a forked prediction stack for GLL. Simple optimizations convert these forked stacks to directed acyclic graphs, or DAGs. Infinitely ambiguous grammars cause these DAGs to degenerate into full-fledged graphs and come with a host of parsing problems (Sections 11.1.3 and 11.2.3). Given the limited usefulness of infinite ambiguity, rejecting such infinitely ambiguous input grammars is a serious option.

To our knowledge, nothing has been published on generalized non-canonical parsing.

Problems

Problem 11.1: Since the LR automaton of Figure 11.2 is already inadequate anyway, we can just as well leave out the terminator `$`; then state 6 disappears and state 3 becomes inadequate too. How does this affect the parsing in Figure 11.3?

Problem 11.2: Modify the GLR algorithm so it produces a parse forest grammar rather than a parse forest. Does the resulting grammar need cleaning?

Problem 11.3: Design a way to allow nullable left-recursive non-terminals in a GLL parser without requiring nullability information from the LL parser generator (Section 11.2.3).

Problem 11.4: *Research project*: Give a formal proof of the GLL parsing algorithm of van Deudekom and Kooiman (Section 11.2) or a provable version of it.

Problem 11.5: *Project*: Investigate how generalized parsing can be done with less usual bottom-up methods: LR-regular, non-canonical. How about using a non-deterministic handle recognizer (as, for example, in Figure 9.15)?

12

Substring Parsing

In Chapter 2 grammars were explained as finite devices for generating infinite sets of strings, and parsing was explained as the mechanism for determining whether a given string — a sentence — belongs to the set of strings — the language — generated by the grammar. As a bonus parsing also reveals the syntactic structure of that sentence.

But sometimes the sentence is not completely known. Parts of it may be missing due to transmission errors, lost in milliseconds in an electronic network or in millennia on clay tablets. It may be syntactically almost but not quite correct, in which case it will consist of a sequence of syntactically correct fragments, "substrings". Or we may have missed its beginning and picked up only somewhere in the middle, in which case we obtain a "suffix" of a correct sentence. In each of these cases we want to do the best syntax analysis possible; so it is useful to be able to do substring and suffix recognition and/or parsing.

In principle we have to distinguish between a substring — a section from a sentence — and a suffix — the tail of a sentence —, but in practice the difference is not that big. Most suffix parsing algorithms are directional, which means that if the suffix happens to be a substring, the algorithm just stops prematurely on end-of-input. In fact, all "substring parsers" mentioned in (Web)Section 18.2.3 are actually suffix parsers.

We do, however, have to distinguish between recognition and parsing. As usual recognition is easier than parsing, and all known algorithms start by recognizing; additional work is then needed to obtain a — necessarily incomplete — parse tree. Suppose we try to parse the string `-n)` as a suffix with the grammar for arithmetic expressions from Figure 9.23, which we repeat here in Figure 12.1. Then a recognizer

```
S  →  E
E  →  E - T
E  →  T
T  →  n
T  →  ( E )
```

Fig. 12.1. The grammar for differences of numbers, copied from Figure 9.23

will just say "Yes"; additionally it will probably provide an incomplete parse tree like the one in Figure 12.2(*a*), but this incomplete tree can be completed in arbitrary ways, each corresponding to a different missing prefix.

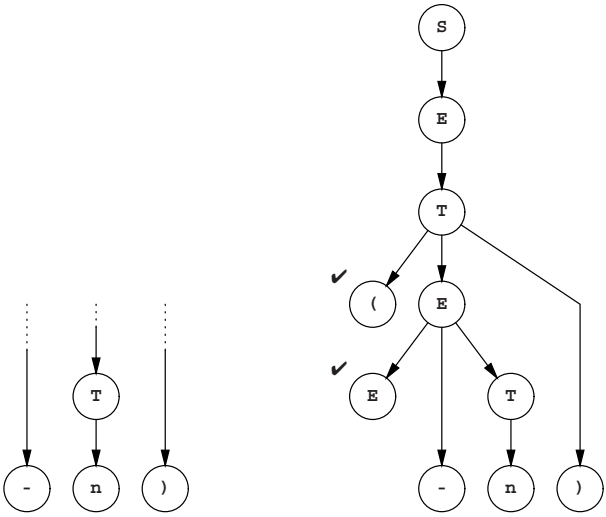


Fig. 12.2. Two kinds of parse structures for the suffix `-n)`

There are two useful data structures that can (usually) be obtained from suffix parsing: a maximally supported parse forest, and a minimally completed parse tree.

The *maximally supported parse forest* is the maximal set of supported parse subtrees, where a supported parse subtree is a node with its subtree all of whose leaves are tokens in the suffix. The maximally supported parse forest is the maximal set of nodes for which there is direct evidence in the input, and for which synthesized attributes can be computed in those cases where semantics must be retrieved (Section 15.3.1). An example is shown in Figure 12.2(*a*), where only the node for `T` is fully supported.

A *minimally completed parse tree* is the smallest tree with a top node labeled with the start symbol of the grammar that includes all subtrees of the maximally supported parse tree plus all input tokens not included in these. It is the smallest tree that gives a complete account of all tokens in the suffix, but it will probably contain non-terminals as leaves. These belong to the prefix, and allow hypotheses about that prefix to be formulated. An example is shown in Figure 12.2(*b*), where the nodes marked with a ✔ are terminals and non-terminals for the prefix that are inferred from the suffix.

It is surprisingly simple to create a grammar for the suffixes of a language L, given a CF grammar for L. Substring parsing based on such grammars has been known for a long time; it is treated in Section 12.1. More efficient techniques for

general substring parsing emerged in the late 1980 and have been extended since (Section 12.2). Techniques for *deterministic* substring parsing followed 5 to 10 years later (Section 12.3). Most of these parsing algorithms are actually recognizers. A few authors report research on constructing completed parse trees; see Lang [210] and Nederhof and Bertsch [216, Section 3].

12.1 The Suffix Grammar

We define the *suffix language* of a language L as the set of strings obtained by removing one of more tokens from the front of sentences in L. In general this suffix language of L contains some strings that also occur in the language L itself. For example, deleting the prefix `n-` from the string `n-n` in the language from Figure 12.1 yields another correct string in that language: `n`. In other words, a string in the suffix language of L is not automatically an erroneous string for L, even tough it *is* a proper suffix.

The suffix language of a context-free language is again a context-free language, and a grammar G_S can be constructed given the grammar G of the original language. Such a grammar is called a *suffix grammar*, and it can be constructed as follows: for every non-terminal A in G, we introduce a new non-terminal A' which derives a suffix of a sentence generated by A. If G contains a rule

$$A \quad \rightarrow \quad X_1 X_2 \cdots X_n$$

the suffix grammar G_S will also contain this rule and in addition it will contain the following series of rules deriving suffixes of what A can derive:

$$\begin{array}{lcl} A' & \rightarrow & X_1' X_2 \cdots X_n \\ A' & \rightarrow & X_2' \cdots X_n \\ \cdots & \cdots & \cdots \\ A' & \rightarrow & X_n' \end{array}$$

A' can be thought of as a "damaged" A, where the damage has been restricted to the front. So if X_i is a terminal symbol, X_i' is the empty string, since that is the result of damaging a terminal symbol. No rule is generated for $A \rightarrow \varepsilon$; see Problem 12.2 for a small complication. If S is the start symbol of the original grammar, the suffix grammar has start symbol S'.

All new non-terminals A' produce the empty string, which is also a suffix, albeit a degenerate one. We can see this as follows. The last rule created for A' is $A' \rightarrow X_n'$. Suppose X_n is a terminal; then X_n' is empty and A' produces empty. Otherwise X_n is a non-terminal, and we consider the last rules created for all production rules of X_n', etc. Somewhere along this chase we must find a Z_{last} which *is* a terminal; otherwise all right-hand sides would contain non-terminals, and the grammar would be non-productive (Section 2.9.3). So the corresponding Z_{last}' is empty and by following the way back we see that all the other non-terminals we met produce empty, including X_n' and A'.

An important consequence of this is that we do not need rules like $A' \rightarrow X_2 \cdots X_n$ to represent a missing X_1. A missing X_1 is represented by X_1' producing empty: the X_1 is completely "damaged away".

When we apply this construction to the grammar of Figure 12.1, we obtain the suffix grammar shown in Figure 12.3. Note that the rule **E' →T** is actually the rule

```
S's  →  E'
E    →  E - T | T
E'   →  E' - T | T | T'
T    →  n | ( E )
T'   →  E ) | E' ) | ε
```

Fig. 12.3. Suffix grammar for the grammar of Figure 12.1

E' →-'T where **-'** (a minus sign with the first symbol removed) is empty; something similar occurs in the rules **E' →E)** due to a deleted **(**, and in **T' →**ε due to a deleted **n**.

Some rules may be produced more than once in this process. For example, the rule **E' →T'** derives both directly from **E→T** and from **E→E-T** by leaving only a suffix of **T**. But since production rules form a set, only one copy is retained.

We see that the suffix grammar is ambiguous. For example, the string **n)** can be produced by **S's →E' →T' →E) →T) →n)** and by **S's →E' →T' →E') →T) →n)**. In the first case it could stem from a damaged **(n)** and in the second case from **(n-n)**. This suggests that it will not be easy to construct an efficient parser for it.

More generally it turns out that suffix grammars almost never have any redeeming properties, and that they can only be handled by general CF parsing. In principle this does solve the suffix parsing problem: construct the suffix grammar and use any general CF parsing algorithm. But the solution is unsatisfying: it is inefficient and does not gives us any insight in the nature of suffix parsing.

More efficient and interesting methods have been found, which use the original grammar (mostly). Some of these have linear time requirements, which makes them very useful for error recovery, and even the non-linear ones are more efficient and less brute-force than the use of suffix grammars.

12.2 General (Non-Linear) Methods

General CF recognizers come in two varieties, non-directional and directional, and so do general suffix recognizers that use the original grammar. The non-directional one is bottom-up and CYK-like; it is described immediately below. The directional one is top-down and is a variation of the Earley algorithm; it is treated in Section 12.2.2.

There is no report in the literature on an Unger-like top-down suffix parser, and it is indeed difficult to imagine how its try-and-divide concept could be exploited for

suffix parsing. But then, some approaches that were thought impossible in the late 1980s have now yielded interesting algorithms, so who knows.

12.2.1 A Non-Directional Method

When we apply the CYK algorithm (Section 4.2) to a suffix rather than to a complete sentence, it will of course fail to identify the start symbol in position (1,*n*) of the recognition table, but it will still identify all substrings that are complete productions of non-terminals. See, for example, Figure 12.4, where we show the result of running CYK on the suffix `n-n)`. The recognition table (*R*) is in tabular format as described

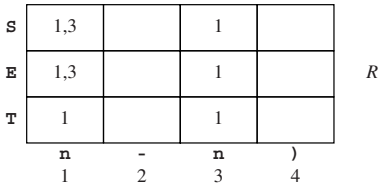


Fig. 12.4. Tabular recognition of the suffix `n-n)`

in Section 4.3, where the vertical axis names the non-terminals and an entry $R_{i,N}$ contains the set of lengths of substrings recognized for N starting at i.

We see that the entry (1,`S`) does not contain the length 4 since `n-n)` is not a correct sentence for the grammar. But all fragments that could be completed have been identified, and can be combined to give various interpretations of the string. Combinations are for example `S)`, since `S` can cover tokens 1...3 and the `)` covers token 4; `E-S)`, since `E` and `S` can cover the tokens 1 and 3 respectively, and the terminal symbols cover themselves; and `E)`, in a way similar to `S)`.

The question arises as to how we can find out if one of these combinations corresponds to a suffix of the grammar. So rather than a sentence start symbol, the presence of which can be checked with a simple test, we need one or more suffix start sequences, which we have to match with combinations of entries in the recognition table.

We will first discuss a way to obtain the suffix start sequences and then how to perform the matching.

12.2.1.1 The Root Set

The suffix start sequences of a grammar G form a language L^{suffix}, but it is an unusual language in that the non-terminals in G occur as terminals in it. For our example grammar it contains strings like `E)`, `-T)))` and ε. Now we have already got a suffix grammar for it, the one in Figure 12.3, but it produces the suffix language expressed in terminal symbols as they appear in the input string rather than in the symbols that appear in the CYK recognition table. This suffix grammar can, however, be adapted

to our needs by removing the rules of the original grammar from it, leaving only the rules for "damaged" non-terminals. **S'** becomes the start symbol and the non-terminals of the original grammar, **E** and **T**, become terminals. The result is shown in Figure 12.5.

```
S'  →  E'
E'  →  E' - T
E'  →  T
E'  →  T'
T'  →  E )
T'  →  E' )
T'  →  ε
```

Fig. 12.5. Root set grammar for the grammar of Figure 12.1

It produces the suffix start sequences **E)**, **-T))))**, ε and many others. Since the start symbol is also called the "root" of a grammar, the language of suffix start sequences is called the *root set*, and its grammar a *root set grammar*.

When we look at the root set grammar, we see something remarkable: the non-terminals occur only in the first position in the right-hand sides, so it is a regular grammar! Actually, the root set grammar of *any* CF grammar is (left-)regular. The reason is that suffix grammar rules for "damaged" non-terminals are defined so that the damage occurs only in the first position, never further on in the rule. (Sometimes the damage is not directly visible, as in the rule **T' →E)**, where the damage done to the leading open parenthesis made it disappear.) This left-regularity makes the matching with the recognition table entries much easier; in fact, it is what makes the algorithm work. If the root set grammar had been a CF grammar, we would have to do CF parsing on the entries in the recognition table. But the finite-state nature of the root set grammar allows matching to be performed efficiently.

12.2.1.2 The Root Automaton

The matching algorithm is based on a deterministic finite-state automaton for the root set grammar. A non-deterministic FSA for our root set is given in Figure 12.6(*a*). We could have constructed it from the root set grammar of Figure 12.5 using the general techniques from Section 5.6, but it is more instructive to derive it directly from the original grammar shown in Figure 12.1. A rule $A \to \alpha$ in the original grammar is represented in the NFA as a sequence of transitions labeled by the symbols in α, starting in an inaccessible state ✘ and ending in a state A'_f for "A finished". This is reasonable since if we have followed some transitions ending up in a state A'_f without starting at the beginning, we have seen a suffix of A.

The transition sequences can be entered from state ε_f after any transition on a terminal; this represents the situation that the suffix starts right after that terminal. It can also be entered from a state X'_f after a transition on the non-terminal X; this

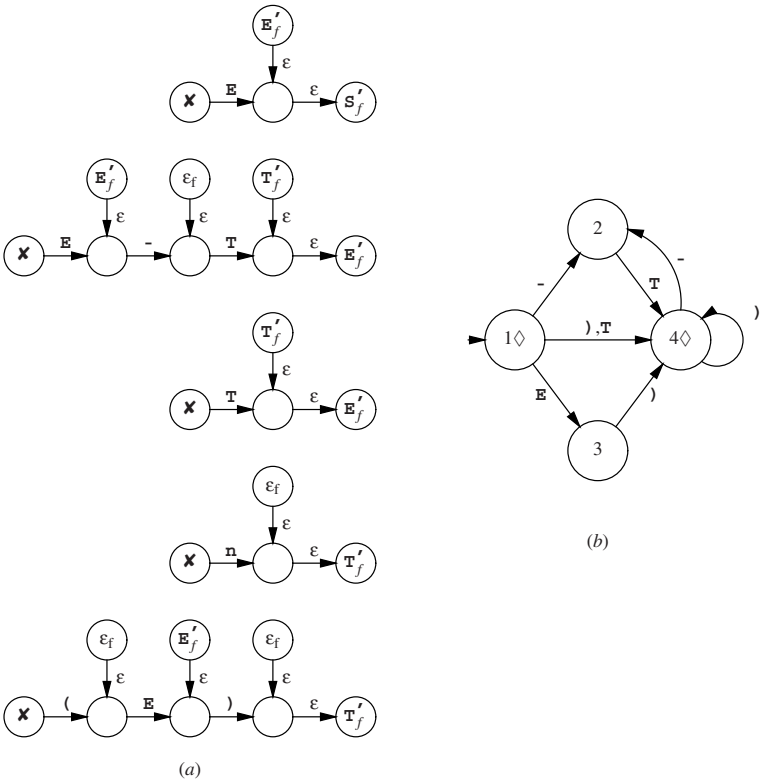


Fig. 12.6. NFA (*a*) and DFA (*b*) for the root set of the grammar in Figure 12.1

happens when the suffix starts with a damaged X. The start state of the automaton is ε_f as it is in any left-regular non-deterministic automaton, and its end state is S'_f, where S is the start symbol of the original grammar.

Applying the subset construction (page 145) turns this root NFA into the root DFA shown in Figure 12.6(*b*). Here state 1 is $\{\varepsilon_f, \mathtt{S}'_f, \mathtt{E}'_f, \mathtt{T}'_f\}$ and state 4 is $\{\mathtt{S}'_f, \mathtt{E}'_f, \mathtt{T}'_f\}$; states 2 and 3 were nameless in the NFA. State 1 is the start state since it contains ε_f, and 1 and 4 are end states since they contain $\mathtt{S}'_f$. It is remarkable that `n` and `(` do not figure in the DFA. The reason is that `n` can always be reduced to `T`, suffix or no suffix; and `(` can only occur in the undamaged right-hand side `(E)`.

12.2.1.3 Matching the Root Set in the Recognition Table

It is easy to see that the root DFA of Figure 12.6(*b*) accepts the combination `E)` available in the recognition table R of Figure 12.4 but not `S)` or `E-S)`, which are also allowed by the table; this is just as we expected.

The algorithm to match members of the root set to the combinations in R can be explained as follows; refer to Figure 12.7, where we record in a table T the states

of the root DFA between the input tokens. This T has $n+1$ elements, numbered

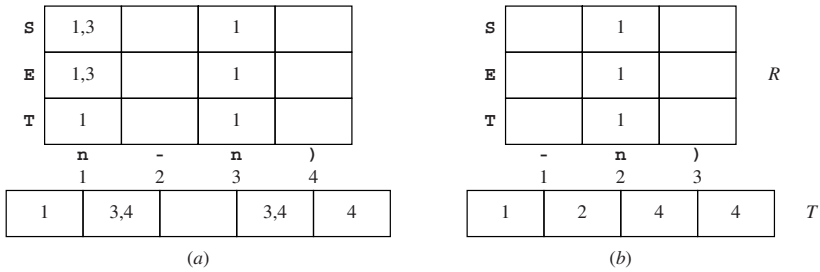


Fig. 12.7. Tabular recognition and root DFA tables for the suffixes `n-n)` and `-n)`

$1 \ldots n+1$. So T_i contains the state set just before the i-th token.

Our root DFA is in state 1 before the first position in the input. From this position it can shift over `-`, `)`, `T`, and `E`. Of these, only `T` and `E` are available in R; let us do $R_{1,\mathrm{E}}$ first. The entry tells us that `E`s of lengths 1 and 3 can start in position 1. First we shift the automaton over the `E` of length 1. It then lands in state 3 between the first and the second token, and we record this in T_2. Then we return to the `E`, shift now over its length 3, and see that the DFA lands in state 3 in T_4. Likewise, a shift over the `T` of length 1, available from $R_{1,\mathrm{T}}$, moves the DFA to state 4 right after the first `n`, in T_2.

So shifts from all root DFA states in front of a given column of R over all grammar symbols and all lengths in that column cause the DFA to jump to various states in various positions between input tokens. Shifts over terminal symbols are also possible; this happens after the second `n`, where the root DFA moves from state 3 over the `)` to state 4 in T_5.

It is clear that we can repeat these actions for all entries of R. In this way we obtain sets of states the root DFA can be in after each token. If the state set right after the last token contains an end state of the root DFA, the input string is a suffix in the language.

The last state set, T_{n+1}, contains the state 4, which is an end state of the root DFA, confirming again that `n-n)` is a suffix. We also see that there are more state sets that contain end states: T_1 at the beginning and T_2 after the first `n`. This means that ε and `n` are also suffixes, but since the input continues after them, they are actually substrings. T_3 does not contain an end state, so `n-` is not recognized as a suffix; indeed it is not. Figure 12.7(*b*) shows that something completely different happens for the suffix `-n)`.

There are a few details we have to look at in this process. The first is that if a length set in an entry of R includes 0, the new DFA state will land in the same entry of T that we are processing. If the resulting state is already there, nothing needs to be done, but if it is not, we need to make sure that it will be processed. Second, we note that since the R entries do not include negative numbers, new states will never

be added to positions in T to the left of where we are working. So if we process the elements of T from left to right, all DFA states will be available when we need them.

The algorithm has quadratic time requirements with respect to the length of the input. This can be seen as follows. There are $O(n)$ entries in T to process. Each can hold $|D|$ states, where $|D|$ is the number of states of the root DFA. For each state we have to process $|V_N|$ non-terminals, each with possibly n lengths (and $|V_T|$ terminals of length 1, but they are negligible). Together this is $O(|D||V_N|n^2)$. Filling the CYK recognition table already costs at least $O(n^3)$, so the additional $O(n^2)$ for suffix matching is negligible.

It is evident that the recognition table R and the matching table T together contain much information for the construction of one or more completed parse trees.

The root set and the matching algorithm are from Bertsch [215].

12.2.2 A Directional Method

Running a CYK parser on a suffix was easy; being a bottom-up parser, it immediately started recognizing fragments without the need for any modification to the algorithm. The hard part was figuring out what to do with the fragments.

Running an Earley parser on a suffix is less easy since the first thing it needs is an initial item set. As we have seen (Section 7.2), the initial item set is computed from the start symbol by the Predictor. This computation is part of the top-down component of the Earley parser, which ensures that all items in all item sets are derivable from the start symbol. But a suffix has no start symbol (unless we use a suffix grammar); a suffix in a language begins at a moment when the start symbol has already been lost.

The easiest way to solve this problem is to just include all possible items in the initial item set; a prophet who predicts everything is never wrong. This immediately raises the question from which position these items originate, what should be filled in for the i in an item like $A \rightarrow \alpha\bullet\beta@i$. We would like to put all possible (non-positive) numbers here, and we do that by writing the joker symbol `*`. This gives the *suffix start set* of the grammar. For the grammar of Figure 12.1 it contains 14 items; example are `S→•E@*`, `E→E•-T@*` and `E→T•@*`. The set is shown as $active_1$ in Figure 12.8. Running the Predictor on $active_1$ yields $predicted_1$.

Since the standard Earley algorithm does not do computations on the origins of items and uses them only in the Completer, it needs remarkably little modification from here on. The shift over the `-` is standard, resulting in zero completed, one active and two predicted items: $itemset_2$. We see that items appear that have known points of origin, for example `E→•T@1` and `T→•n@2`. The shift over the `n` is survived by only one item, `T→•n@2`, which turns into `T→n•@2` and lands in $completed_3$. The Completer goes to the position of origin, 2, finds there all items that have the dot in front of a `T`, of which there is only one, `E→E-•T@*`, and turns it into `E→E-T•@*`.

Finding an item `E→E-T•@*` in $completed_3$ means that an `E` has been recognized starting somewhere before the left end of the suffix and ending here. Now the Completer cannot go to an $itemset_*$ because it would be situated before the beginning of the suffix. This is solved in the same way as the creation of the initial item

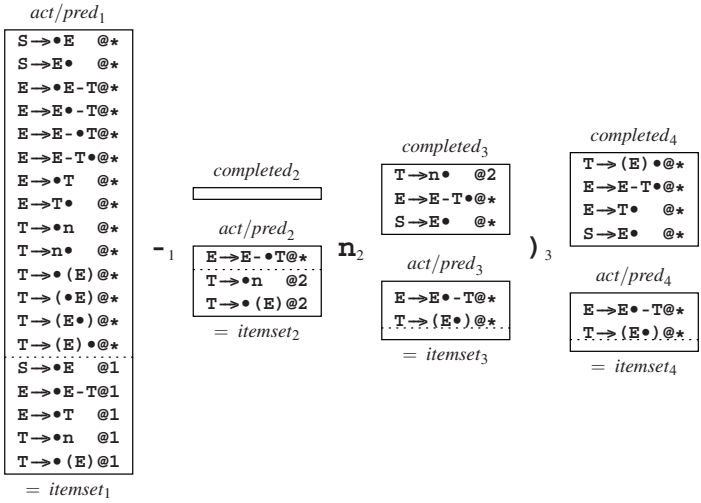


Fig. 12.8. Earley suffix parsing of `-n)`

set: assume that $itemset_*$ contains all possible items; so it is actually equal to $active_1$. This is another modification that needs to be made to the original Earley parser to turn it into an *Earley suffix parser*.

Filtering the items from there that have the dot in front of an `E` yields `S→E•@*` (which goes to $completed_3$) and `E→E•-T@*`and `T→(E•)@*` (which go to $active_3$). There is nothing to predict here. Similar considerations allow us to construct $itemset_4$.

A lot of information can be gathered from inspecting the items sets at the end of the input. Since $itemset_4$ contains the item `S→E•@*`, we have found that `-n)` is indeed a suffix, but we can also see that it is not a complete sentence. If it were, we would also have found an item `S→E•@1`. Additionally, since $itemset_3$ contains the item `S→E•@*` too, `-n` is a suffix as well. The $itemset_4$ shows that the input could continue with a `-`, leading to another `-T`, or with a `)` deriving from another `(E` in the missing prefix.

The time requirements of the Earley suffix parser are the same as those of the standard Earley parser: $O(n^3)$; the size of the initial item set has no influence. The Earley suffix parser is described in more detail by Nederhof and Bertsch [216, Section 3].

Rekers and Koorn [212] have described a GLR parser modified to do substring parsing.

12.3 Linear-Time Methods for LL and LR Grammars

It had been known since the mid-1960s that LL and LR languages could be parsed in linear time, but for decades there was no way to do the same for LL or LR suffix

languages. The suffix grammar of an LL(1) grammar is definitely not LL(k), and the same is true for LR grammars; also, neither seemed to have any other exploitable qualities. So the only way to do LL or LR suffix parsing was through general, cubic-time, parsing. This was bothersome, especially in error recovery, where suffix languages are important (Section 16.7).

A beginning of a break-through occurred when in 1993 van Deudekom and Kooiman [170] implemented a generalized deterministic top-down parser, applied it to suffix grammars of LL(1) grammars, and found to their amazement that they could not make the parser require cubic time even if they tried. What apparently happened was that already after a few tokens the parser had constructed so much of a prediction stack that it could run in linear time for quite a distance, and only at increasingly wider intervals did it have to resort to wild guessing.

The real break-through came one year later, when the situation was analysed by Bertsch, who reported a provably linear-time LL(1) suffix parser [215]. Several other linear-time suffix parsing methods for LL and LR languages were then discovered. They come in two varieties: linear versions of general methods (Section 12.3.1 and 12.3.2) and tabular methods (Section 12.3.3).

An algorithm for linear-time LR suffix parsing was developed independently by Bates and Lavie [214] in 1980 but was not published until in 1994; it is covered in Section 12.3.2.

A linear-time suffix parsing algorithm for BC(1,1) grammars was reported by Cormack [211] in 1989.

12.3.1 Linear-Time Suffix Parsing for LL(1) Grammars

Surprisingly, and perhaps disappointingly, we do not need a new algorithm to do linear-time suffix parsing for LL(1) grammars. It turns out that an Earley suffix parser with a 1-token look-ahead works in linear time when used with an LL(1) grammar. We shall first show that this property holds for the standard Earley parser and then that it also hold for the Earley suffix parser.

12.3.1.1 The Earley-on-LL(1) Parser

Intuitively it is probably not too amazing that an Earley-on-LL(1) parser has better than cubic time requirements, but showing that it works in linear time in the length of the input is not easy. Before we embark on that, we show a run of an Earley parser on an LL(1) grammar. For this we use an LL(1) version of the grammar for differences of numbers used above. It was constructed by removing the left recursion from the original grammar in Figure 12.1 using the technique from Section 6.4 and touching up the result a bit by hand. It is shown in Figure 12.9; $\mathbf{E}_\mathrm{t}$ stands for $\mathbf{E}_\mathrm{tails}$.

To allow the Earley parser the full benefit of the LL(1) parse table we modify it so its Predictor will base its decisions on that table rather than on FIRST sets as it did in Section 7.2.4.1. So when predicting the items for an item $A \rightarrow \cdots \bullet B \cdots @i$ where the next input token is t, it uses the entry (B,t) of the LL(1) parse table to predict an item $B \rightarrow \bullet \cdots @\, j$. The parse table is

```
E   →  T Et
Et  →  - T Et
Et  →  ε
T   →  ( E )
T   →  n
```

Fig. 12.9. LL(1) grammar for differences of numbers

	n	-	(	)	#
E	TE_t		TE_t		
E_t		$-TE_t$		ε	ε
T	n		(E)		

where we have used FOLLOW(E_t)={),#} to get the correct predictions for the rule $E_t \rightarrow \varepsilon$.

We now use this parser to parse the string **(n-n)**. The process and the item sets are shown in Figure 12.10. Since we are dealing with a parser which uses a one-token

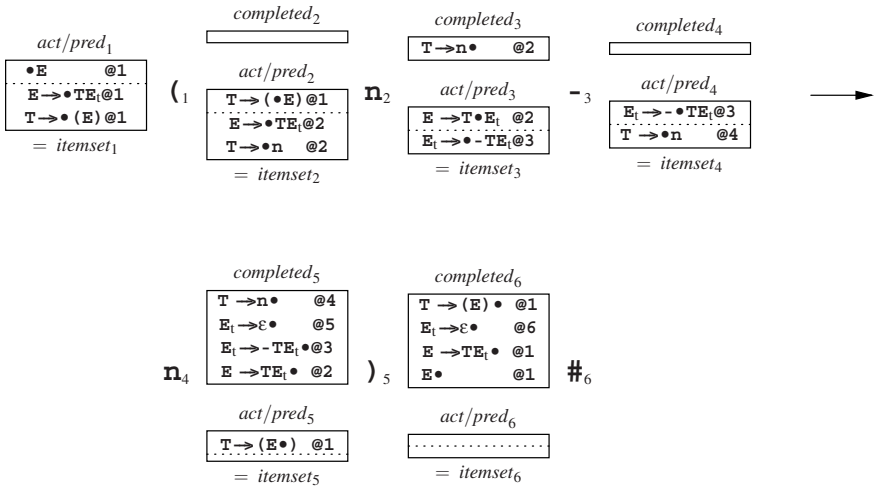


Fig. 12.10. Earley-on-LL(1) parsing of the string **(n-n)**

look-ahead we have extended the input with one **#**. We have taken the station **•E** as the start item; consequently we observe that the whole string has been recognized by finding the item **E•@1** in $itemset_6$. Note that using the LL(1) parse table and having the end-of-input token **#** together prevent any further predictions from being made.

12.3.1.2 Properties of the Earley-on-LL(1) Parser

We shall first prove two properties which hold for a slightly modified Earley on LL(1) parser, use these properties to show linearity, and then show that the modification

makes no difference to the linearity. We will discuss the modification when we need it. The properties are:

P1. The set $active_i$ holds exactly 1 item for all $1 \leq i \leq n$.
P2. The set $itemset_i$ holds at most 1 item of the form $A \rightarrow \cdots \bullet X \cdots @\, j$ for any terminal or non-terminal X.

Property *P1* certainly holds for $active_1$: the one active item is the start item. If the dot in the active item is in front of a terminal, the Predictor does not add anything, and *P2* holds for $itemset_1$. If the dot is in front of a non-terminal, say A, the Predictor looks at the first token in the input, t_1, and consults the entry (A,t_1) of the LL(1) parse table. Since the grammar is LL(1) it will find there at most one rule; suppose that rule is $A \rightarrow B \cdots$. It then turns this rule into the item $A \rightarrow \bullet B \cdots @1$ and inserts it in $predicted_1$. It is important to note that B cannot be the same as A, since otherwise A would be directly left-recursive and the grammar would not be LL(1). If B is a non-terminal, the Predictor now repeats the process for B, resulting in an item like $B \rightarrow \bullet C \cdots @1$. Here C must be different from A and B, or the grammar would be left-recursive. The process is repeated until an item appears in which the dot is in front of a terminal. All non-terminals after the dot are different and there is exactly one item with a terminal symbol after the dot. So property *P2* holds for $itemset_1$. For the moment we assume that there are no ε-rules in the grammar.

Next the Scanner moves the item(s) that contain $\bullet t_1$ over the token t_1, and we know from *P2* that there is only one such item. If it is not a completed item, it is entered into $active_2$; so property *P1* holds at position 2. Then the Predictor does its work as above, establishing *P2*. If it *is* a completed item, it is entered into $completed_2$, and the Completer goes to work. Say the completed item is $P \rightarrow \cdots \bullet @1$. The Completer then goes to $itemset_1$, and finds the item(s) in it that contain $\bullet P$. Since this is an Earley parser there will be at least one such item, and since *P2* holds at position 1, there is at most one such item, so there will be exactly one such item. The Scanner in the Completer turns the $\bullet P$ into $P\bullet$. The resulting item can again be completed or not completed, and the process is repeated until a non-completed item is obtained (unless P is the start symbol; see below), which then duly goes into $active_2$. So again property *P1* holds. And again the Predictor establishes *P2*.

The argument given above for position 2 can be repeated for all further positions, since for each position i the argument only assumes that the properties *P1* and *P2* hold for all positions $< i$. So we can conclude that *P1* and *P2* hold for $1 \leq i \leq n$.

There are two complications here. A minor one is that the Completer/Scanner loop does not come up with a non-completed item just after the last input token; there the last obtained item is the completed start symbol, which goes into $completed_{n+1}$. So *P1* does not hold for position $n+1$, as we can see in Figure 12.10.

The second complication concerns ε-rules. When the Predictor in position i predicts an ε-item, say $C \rightarrow \bullet @\, i$, from an active item, say $A \rightarrow B \bullet CDE @\, j$, the predicted item is immediately processed by the Completer, which combines it with the item $A \rightarrow B \bullet CDE @\, j$ to produce $A \rightarrow BC \bullet DE @\, j$, again an active item. So now there are two items in $active_i$, violating property *P1*. But a little thought reveals that once the item $A \rightarrow BC \bullet DE @\, j$ has been introduced, the old item $A \rightarrow B \bullet CDE @\, j$ has become

useless. The reason is that only one item of the form $C \rightarrow \bullet \cdots @i$ has been generated for the $\bullet C$ due to the LL(1) property, and it has been consumed immediately. No other item of the form $C \rightarrow \bullet \cdots @i$ will ever exist, and the item $A \rightarrow B \bullet CDE @ j$ will never be used again. So we modify the Earley-on-LL(1) parser to discard such items. Although the process may repeat itself for E, etc, once it stops, it yields either an active item that does not predict an ε-item, in which case *P1* holds, or a completed item, which is then again processed normally. So, although a lot may happen in the Completer/Scanner/Predictor loop, in the end *P1* and *P2* hold. Note that this situation does not occur in Figure 12.10, since all ε-item processing results in completed items.

12.3.1.3 The Linearity of the Earley-on-LL(1) Parser

We will now show that the thus modified Earley-on-LL(1) parser runs in linear time, and we will do this by showing that at each position the Scanner and the Predictor use an amount of time independent of the length of the input, n, and that the Completers in all positions together do an amount of work proportional to n.

Since there is only one item with $\bullet t_i$ in $itemset_i$, the Scanner has to process and produce one item only. The Predictor starts from exactly one item (property *P2*). It may produce more than one item, but since all the non-terminals following the dot are different (property *P2*) their number is limited by the number of non-terminals in the grammar.

The Completer may often do nothing, but when it is called it may do lots of consecutive reductions. When it is asked to process a completed item $B \rightarrow \cdots \bullet @ j$ in position i, it goes to $itemset_j$, selects the one item with $\bullet B$ (property *P2*), say $A \rightarrow \cdots \bullet B \cdots @k$, and completes it to $A \rightarrow \cdots B \bullet \cdots @k$. This new item may again be a completed item, etc., and it is difficult to assess the total cost. But it is interesting to see how the item $B \rightarrow \cdots \bullet @ j$ ended up in position i. It came from one item $B \rightarrow \cdots \bullet \cdot @ j$ in an earlier position, which came from an item $B \rightarrow \cdot \bullet \cdots @ j$ in an even earlier position, etc., which finally came from an item $B \rightarrow \bullet \cdots @ j$ in position j. So there is a one-on-one unbroken chain from the item $B \rightarrow \bullet \cdots @ j$ in position j to the item $B \rightarrow \cdots \bullet @ j$ in position i. This means that now that the item $B \rightarrow \bullet \cdots @ j$ in position j has been used, the Completer will never use it again, for lack of further clients for that item. And since the Completer will touch each item in all *itemset*s only once, and since we have just seen that there are only $O(n)$ of these, all Completers together do an amount of work proportional to n.

This shows that the modified Earley-on-LL(1) parser works in linear time. But the modification is not essential to the linear-time argument, which we can see as follows. Suppose we leave the used-up items of the form $A \rightarrow B \bullet CDE @ j$ in. Then they will never be touched again by the parser and cannot influence its time dependency; so we might as well leave them in. This shows that the unmodified Earley-on-LL(1) parser also runs in linear time.

12.3.1.4 The Earley-on-LL(1) Suffix Parser

It is quite simple to use this Earley-on-LL(1) parser to implement a linear-time Earley-on-LL(1) suffix parser. It is sufficient to just take each item in the suffix start set of the grammar, and run an Earley-on-LL(1) parser with that item as the only item in $active_1$. If one of these parsers succeeds, the input is a suffix. Each parser is linear, and since the number of parsers depends only on the grammar and not on the length of the input, the combined, sequential, parser runs in linear time. This sounds wasteful, but it may not be: most parsers will fail immediately or after a few tokens. But the overhead is high and we agree that it is not elegant, so a better approach is needed: process all items in the suffix start set simultaneously by entering all of them in $active_1$, just as in the Earley suffix parser in Section 12.2.2.

Figure 12.11 shows such a parser at work on the suffix `-n)`. The struck out items

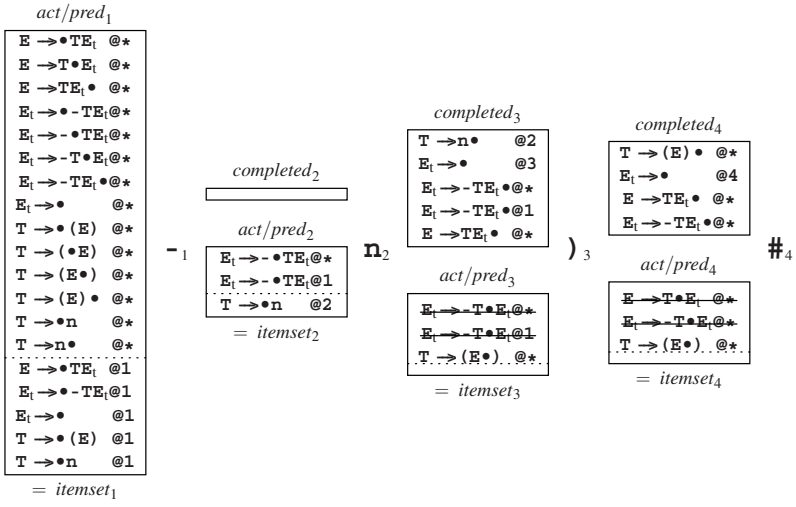


Fig. 12.11. Earley-on-LL(1) suffix parsing of `-n)`

in $active_3$ and $active_4$ are items that were, or could be, discarded as being no longer useful because an ε-prediction ($\mathtt{E_t} \rightarrow \varepsilon$) was made from them.

We will now look at the time requirements of this parser. Initially the only difference is that there are more items in the active sets, and property *P1* does not hold. This is no problem for the Scanner and the Predictor themselves. If there are k items they just do k times as much work, and provided k is limited by a constant and does not grow with n, linearity is not threatened. But as a result of their work, several items with the same symbol after the dot can arise in the same item set, so property *P2* no longer holds either. This then causes problems with the Completer, when it processes a completed item $P \rightarrow \cdots \bullet @i$. It can now find several items in $itemset_i$ that have $\bullet P$, so several items will be added to the present sets, and it is difficult to see if they are limited to a constant number.

The Completer can introduce two forms of items, with @$*$ and with @j for some number j. Neither can threaten the linearity of the parser; we can see this as follows. The number of items with @$*$ is limited by the grammar, so the number added by the Completer is also limited, and items with @$*$ cannot destroy the linearity. An item i with @j for some j derives from some item predicted earlier in the parser. But the same item must have been predicted in one of the sequential parsers described in the previous section. So whatever happens further on with I also happened to its counterpart in the sequential parser. That parser was linear, so all its subprocesses were linear or less, so the processing of item I will be linear or less, and cannot destroy the linearity of our parser.

Remarkably, showing that the Earley-on-LL(1) algorithm runs in linear time has been more work than explaining the algorithm itself. A formal (and much more compact!) proof can be found in the paper by Nederhof and Bertsch [216, Section 3].

12.3.2 Linear-Time Suffix Parsing for LR(1) Grammars

Nature is symmetrical; kind of, that is. Just as the Earley parser, modified to do suffix parsing and using an LL(1) parse table runs in linear time, the GLR parser, modified to do suffix parsing and using an LR(1) parse table runs in linear time. But the symmetry is not perfect. Showing that the standard Earley parser runs in linear time when it uses an LL(1) grammar is not simple. But a GLR parser using an LR(1) table is obviously linear: it is just a normal LR(1) parser in which the stack in implemented as a graph, which then degenerates into a linked list.

12.3.2.1 GLR Suffix Parsing

To better see the implications of using an deterministic LR(1) parse table for GLR parsing we consider a graph-structured stack like the one in Figure 11.6 on page 387. If we use an LR(1) automaton with no inadequate states, each state leads to a unique decision, and there will be no forks. This means that if we start from the start state, there is only one path to any point in the input; an example is shown in Figure 12.12(a). Note that the forks in the decision process show up as joins or reverse forks in the diagram, since the arrows point in the direction of the reductions rather than the decisions.

We can simulate what happens when we try suffix parsing by cutting off the left end of the picture in Figure 11.6, and then remove all reverse forks. A possible result is shown in Figure 12.12(b); we see that the graph-structured stack simplifies to a *forest of tree-structured stacks*. And that is indeed the data structure employed by the *GLR suffix parser* designed by Bates and Lavie [214]. (A similar structure was used much earlier by Lindstrom [329].)

Figure 12.13 shows the GLR suffix parser in action. We use the grammar from Figure 12.1; the LR(1) parse table is shown in Figure 9.28 on page 294. The input is again `-n)`, extended with one `#`.

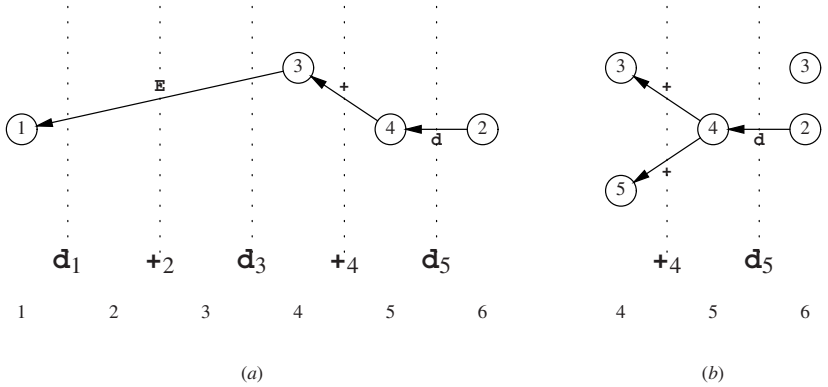


Fig. 12.12. Graph-structured stacks when using an LR(1) parse table

Initially we know nothing about the stacks and as usual we express that lack of knowledge by allowing all possibilities. This gives a forest of 16 empty tree-structured stacks, of which only the roots with states 1 to 4 and 6 to 17 [1] exist; see Figure 12.13(*a*), where the input `-n)` is shown on the right of the frame. One can imagine that to the left of each root with state *s* there are all possible stacks that have state *s* on top, but since there are infinitely many of these, we cannot draw them there. This image is occasionally useful for understanding some optimizations Bates and Lavie describe.

Now we consult the ACTION table to see how each of these states reacts to the look-ahead `-`. Five states, 1, 6, 7, 13, and 14, are erroneous with this look-ahead, so their trees are discarded. Eight states, 2, 3, 8–10, 12, 15, and 17, require a reduction, but there is nothing to reduce yet, so these are also discarded. The other three states 4, 11, and 16, order a shift, which we can do; see frame *b*.

The standard GLR parser combines equal stack suffixes (Section 11.1.2.2), and so does the suffix parser. This combines the two top states 14, leading to our first forest, in frame *c*. The actions prescribed by the ACTION table for look-ahead `n` are shown below the top states: shift and shift. Frame *d* shows the results of the shifts. Since state 3 does not allow a look-ahead `)`, one tree is discarded; state 10 requires a reduction with `T→n`, which saves the other tree (*e*). Here state 15 requires a reduction with `E→E-T`, which causes a problem, since only the tail of the right-hand side, `-T`, is available. This is to be expected in a suffix parser, and we do the reduction anyway. So we land in the cloud of all possible states to the left of the trees, and from there only select those states that have a transition on `E`, the result of our reduction. This is comparable to the selection process going on between frames *a* and *b*. Such a reduction is called a *long reduction*. The result is shown in frame *f*.

Two trees from frame *f* survive the look-ahead test and undergo shifting (*g*). Here one tree survives and the top state 12 with look-ahead `#` requires reduction with `T→(E)`. Since only `E)` is available, this is again a long reduction. There are

[1] State 5 is missing from the table, as explained on page 290.

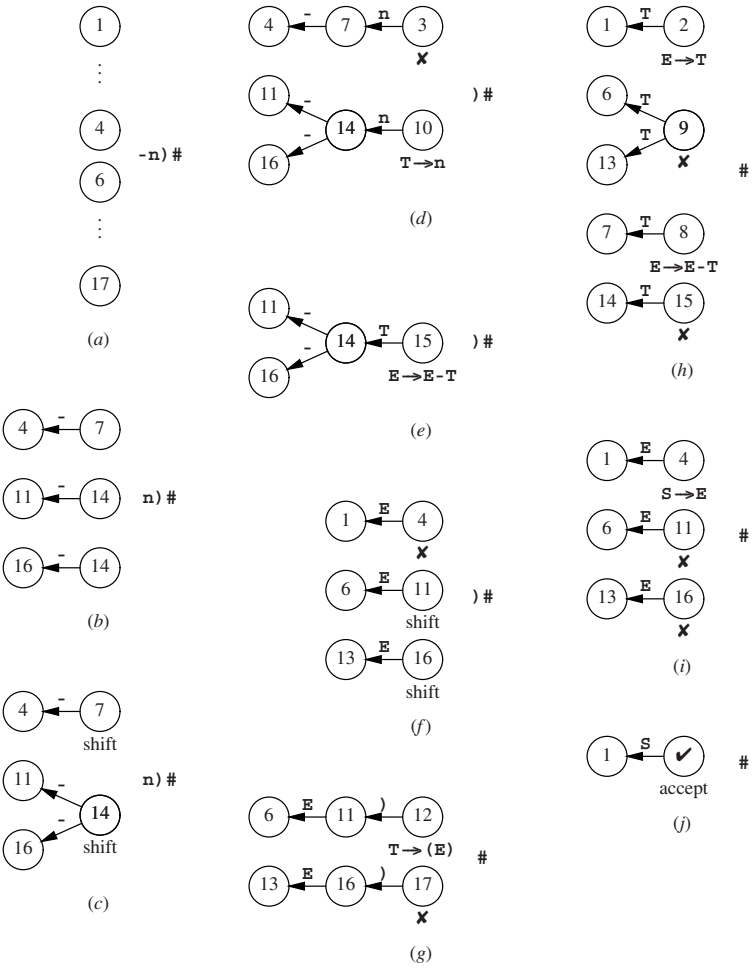


Fig. 12.13. Snapshots of the GLR suffix parser in action on the input `-n)`

more states that have a transition on **T** than on **E**, and four trees result (*h*). The first tree needs a reduction with **E→T**, which is possible and which yields the first tree in frame *i*. The third tree in frame *h* again causes a long reduction, plus a shift on **E**. This results in the same three trees as in frame *f* and these are introduced into frame *i*. This means that the tree ①**E**④ occurs twice; just as in the standard GLR parser, first the top states ④ are merged, and then further branches are merged. Since both trees are equal only one remains (frame *i*). A simple step then leads to frame *j*, where we see that the suffix **-n)** is accepted.

The GLR suffix parser differs only in two points from the standard GLR parser: it starts in all states of the parse table, and when a stack in a tree causes a long reduction $A \to \cdots$, the subsequent shift over A starts from all states.

12.3.2.2 The Linearity of GLR Suffix Parsing

We will now turn to the time requirements of the GLR suffix parser. First we notice that there can never be more than N trees, where N is the number of states of the LR(1) automaton. The tree-combining part of the algorithm makes sure of that, since all trees with the same state on top are combined into one tree. Next we will show that the parser, during its work at a given position i, cannot add more than a constant number c of branches to the trees, where c depends on the grammar only. (It can *remove* any number of branches though.) We then use these properties of the algorithm to show that it runs in linear time.

Working at position i where there are p trees ($p \leq N$), the algorithm can do at most p shifts, which can add at most p branches. As usual, the reduce part is more complicated, as we need to distinguish two kinds of reduce actions: ε-reductions and non-ε-reductions. The first kind adds branches, the second does not, so we can forget about the non-ε-reductions here. Sequences of consecutive ε-reductions can occur, but all have to be different, or the original LR(1) parser would have looped. And since there are only a fixed number q of ε-rules in a grammar, there can only be q consecutive ε-reductions, adding q branches. This can happen at most to all p trees, together adding at most $p \times q$ branches, which still is a grammar-determined constant. The subsequent tree-combining step does not add branches (on the contrary, usually it removes some).

Since the work at each position adds at most c branches, we can conclude that the number of branches in the tree forest cannot be more than $c \times i$ when we are done with position i. So our data will grow at most linearly with the length of the input.

We now consider the time used per position by each of the components of the algorithm. Shifting can cost at most p actions, which is constant. For the reduce part we need to distinguish three kinds of reduce actions: ε-reductions, unit reductions, and "proper" reductions, reductions that actually reduce the number of branches. We have seen that there can be at most a fixed number of ε-reductions per position, costing a fixed number of actions. Unit reductions in a sequence have to be unique too, or there would be a loop in reductions; so the same applies to them. Proper reductions are different. They can occur in unbounded sequences, possibly up to the beginning of the input, and cost a constant amount per branch; they do, however, remove that branch while processing it. The tree combining step stops as soon as it sees two branches it cannot combine, so its cost is proportional to the number of branches removed. In total each step costs a constant amount plus an amount proportional to the number of branches it removes.

When we reach the end of the input of length n, at most $C \times n$ branches have been added to the tree forest, where C is the cumulative constant of all the constant additions. The n steps have cost $F \times n$, where F is the fixed amount per step, plus an amount proportional to the number of branches removed by reducing and tree combining. But that number is limited to $C \times n$, since there were at most that many branches. This adds up to $(F + C) \times n$, which shows that the GLR suffix parser requires linear time.

The GLR suffix parser is a practical and efficient algorithm, which has been applied in real-world software. It works with any kind of deterministic bottom-up table: LALR(1), SLR(1), LR(0), precedence, etc. Full algorithms and correctness proofs are given by Bates and Lavie [214]. Goeman [218] supplies rigorous proof of linearity.

12.3.3 Tabular Methods

The root matching algorithm of Section 12.2.1.3 has quadratic time requirements. The reason is that an entry $R_{A,i}$ in a recognition table (for example Figure 12.4) can contain the lengths $0 \ldots n-i+1$, so the amount of work per position depends (linearly) on n. If this were not so and each entry contained only at most a fixed number of lengths, the root matching algorithm would require linear time. This suggests that if the language is such that each entry contains at most a fixed number of lengths and if we can fill in the recognition table in linear time, we have obtained a linear-time suffix parsing algorithm. This line of thought has been pursued by Bertsch and Nederhof in two papers [215, 216] concerning the LL(1) and LR(1) languages.

12.3.3.1 Filling the Recognition Table for an LL(1) Language

The LL(1) languages are good candidates for filling in the recognition table in linear time. It is not difficult to see that the entries in a recognition table filled in for an LL(1) language contain at most one length each, as follows. One way to fill in the entry $R_{A,i}$ in such a table would be to start an LL(1) parser in position i with a prediction stack which just holds A. This parser will then recognize either one terminal production of A in the segment $i \cdots$ or none. Since it is deterministic it cannot recognize more than one occurrence, so at most one length will be entered in $R_{A,i}$. Although this would work, starting an LL(1) parser for each entry in the recognition table would be inefficient, and a linear-time algorithm is needed.

We will again use the LL(1) grammar from Figure 12.9, and the suffix `n-n)`. The recognition table is shown in Figure 12.14. It has three rows, for **E**, $\mathbf{E}_\mathrm{t}$ and **T**, and five columns, four for the input tokens and one for the appended look-ahead **#**. The idea is to fill in the table from right to left and from top to bottom, guided by the

E	3		1		
$\mathbf{E}_\mathrm{t}$		2		0	0
T	1		1		
	n	**-**	**n**	**)**	**#**
	1	2	3	4	

1	4	2	3,4	4

Fig. 12.14. Tabular LL(1) recognition and root DFA table for the suffix `n-n)`

LL(1) parse table and resolving subproblems in top-down fashion; we will see that this approach is advantageous.

Starting at entry $R_{\mathrm{E},5}$, we are asked to identify an **E** with a look-ahead of **#**; the parse table on page 410 shows us that there is none. For entry $R_{\mathrm{E_t},5}$ we find the prediction $\mathbf{E_t} \rightarrow \varepsilon$, so we can identify an $\mathbf{E_t}$ of length 0 here. Entry $R_{\mathrm{T},5}$ again remains empty.

Column 4 has **)** as look-ahead, and since the predictions for look-ahead **)** are the same as those for **#** (see parse table) we get the same entries in column 4 as in column 5.

Turning to column 3 and entry $R_{\mathrm{E},3}$ we find the prediction $\mathbf{E} \rightarrow \mathbf{T}\mathbf{E_t}$ for look-ahead **n**. This brings us to a subproblem, since the entry $R_{\mathrm{T},3}$, being lower in the tables, has not been computed yet. So we stack the computation of $R_{\mathrm{E},3}$ in top-down fashion, and proceed with $R_{\mathrm{T},3}$, for which we find a prediction $\mathbf{T} \rightarrow \mathbf{n}$ and thus a length of 1. Now we can complete $R_{\mathrm{E},3}$ as $R_{\mathrm{T},3} + R_{\mathrm{E_t},4} = 1 + 0 = 1$. Continuing this way we can complete the entire table. For example, $R_{\mathrm{E_t},2}$ yields the prediction $\mathbf{E_t} \rightarrow \mathbf{-}\,\mathbf{T}\mathbf{E_t}$ and is computed as $R_{\mathrm{E_t},2} = 1 + R_{\mathrm{T},3} + R_{\mathrm{E_t},4} = 1 + 1 + 0 = 2$, where the first 1 is the length of the **-**.

Two things have to be noted here. The first is that to prevent recomputation of entries like $R_{\mathrm{T},3}$ which are later in the table but have already been evaluated as subproblems, we need to mark entries as "done", even if they do not contain a length.

The second is that we can now see more clearly why each entry contains at most one length: 1. the LL(1) property guarantees that there will be at most one prediction for the entry; 2. the components of the right-hand side of a prediction correspond to entries in the recognition table that have already been computed, and if each entry contains at most one length, the resulting entry will also contain at most one length; 3. the lengths that are not computed from components result from ε and unit rules, which identify exactly one length; So we can conclude that all entries will contain at most one length.

The work done per column is determined by the lengths of the right-hand sides of the predictions only and does not depend on the length of the input, so the total task has linear time requirements.

To complete the algorithm we have to run the root automaton, to see if an element of the root set can be recognized in the table. Rather than constructing the root DFA for the grammar of Figure 12.9 as a parser would, we take a short-cut here, and realize that Figures 12.1 and 12.9 describe the same language and that all non-terminals from 12.1 appear unharmed in 12.9. This allows us to just use the root DFA of the original grammar shown in Figure 12.6. So the root set matching table T in Figure 12.14 is filled in on the basis of this automaton. We see that T_5 contains the end state 4, so the suffix **n-n)** is recognized.

The algorithm and its optimizations are described in detail by Bertsch [215].

12.3.3.2 Tabular Suffix Recognition for an LR Language

The technique used for LL(1) languages cannot be adapted easily to LR languages, unfortunately. A working algorithm is easily obtained — it just reverts to general

tabular parsing as explained in Section 4.3 — but it has at least quadratic time requirements. This is caused by several factors. The first is that LR parsers do not start "with a prediction" but "in a state". This can be overcome by indexing the table by states rather than by non-terminals, but the root matching automaton still wants to see non-terminals.

What is worse, two or more subtrees for the same non-terminal can start in the same position in an LR language. For example, the suffix **n-n)** contains two occurrences of **E** in position 1, one of length 1 and one of length 3 (again using the by now slightly overworked grammar of Figure 12.1 on page 399). And an input **n -n**^{k} **)** will contain $k+1$ occurrences of **E** in position 1; so the number of lengths in an entry in the recognition table depends linearly on the length of the input, which causes quadratic behavior.

Equally badly, filling the table from right to left no longer works. Suppose we have a grammar **L→La | a**, which produces **a**$^+$, and a parse tree P_3 for the last 3 tokens in **aaaa**. Now we take one step to the left and want to create a parser tree P_4 for the full **aaaa**. Then we see that the P_4 does not reuse any of the nodes of P_3, because all nodes hinge on the position in which the parse tree starts. So having P_3 does not make getting P_4 any cheaper, and the linearity of Section 12.3.3.1, based on that phenomenon, is lost. Working from left to right solves this case but then a grammar like **L→aL | a** causes problems.

In their groundbreaking paper, Nederhof and Bertsch give an algorithm for linear-time LR(1) suffix parsing [216, Section 4], but it is presented in a framework of push-down automata (PDAs, Section 6.2) rather than of context-free grammars. Although PDAs are fundamentally equivalent to CF grammars, the relationship is so remote that proofs and algorithms from one framework cannot be easily translated into the other framework.

Nederhof and Bertsch's algorithm can be summarized as follows.

- The language to be used is specified by a deterministic PDA (which is a recognition device) rather than by a CF grammar (which is a generative device). The set of transitions allowed in this PDA is restricted. There are three forms: $q \xrightarrow{a} qq_1$, which pushes an a on the stack, packed as state q_1; $q \xrightarrow{\varepsilon} qq_1$, which pushes a new state q_1 on top of q; and $q_1q_2 \xrightarrow{\varepsilon} q_3$, which reduces the state pair q_1q_2 to q_3. It is possible to obtain such a restricted deterministic PDA (*RDPDA*) from an LR(1) grammar; see below.
- This RDPDA still suffers from the same non-linearity problems mentioned above. The problem is identified to arise from "loops" in the RDPDA, and a transformation τ on the RDPDA instructions is defined. It introduces hierarchies of stack symbols, which are used to cut the loops.
- An interpreter similar to the one used in the previous section for the LL(1) suffix recognizer is defined. It combines the filling of the recognition table with the root matching automaton and is proved to have linear time requirements.
- The recognition table is then analysed to retrieve a parse forest [216, Section 6].

Nederhof and Satta [174] explain how to obtain an RDPDA for a given LR(0) grammar. For each LR(0) shift q_1aq_2 we have a transition $q_1 \xrightarrow{a} q_1q_2$, and for each

LR(0) reduce $qAq_1B\cdots q_k$, which identifies a reduction using $P \rightarrow AB\cdots$ in state q_k and where a shift of q over the resulting P yields the state q_p, we have a transition $qq_1\cdots q_k \xrightarrow{\varepsilon} qq_p$. The grammar symbols are not stored on the PDA stack.

The transitions created for the reduce actions of the LR(0) parser do not fulfill the requirements for an RDPDA since some of them remove more than two states from the stack. For each such transition a series of transitions is introduced that removes the states piecemeal from the stack, using new states to keep track on where the process is, in a technique similar to the creation of Chomsky Normal Form in Section 4.2.2.

Nederhof and Satta [174] do not explain how to obtain an RDPDA for an LR(1) grammar, that is, how to handle the look-ahead token. One way to do this is by incorporating the look-ahead symbol in the top state, so one more token of the input has been consumed than would intuitively be assumed. A problem with this is that when states get pushed deeper into the stack, the look-aheads they contain become obsolete, and more transitions are needed to update the look-ahead when these states resurface.

When all these transformations are combined, a very efficient LR(1) suffix parser of great sophistication and complexity results. But it needs a lot of work to make it applicable (see, however, Problem 12.9).

Bertsch and Nederhof [96] use related techniques to obtain a linear-time parser for regular sequences of LR(0) languages.

12.3.4 Discussion

Two linear-time LL(1) suffix parsers have been presented, one based on Earley's algorithm and one based on tabular parsing. Neither is particularly easy to implement; the Earley one uses the LL(1) table only, but the tabular one requires an additional root matching automaton, not related to the LL(1)-ness of the grammar. The Earley suffix parser has another advantage. Most so-called LL(1) grammars are not completely LL(1) in practice, where their defects are handled by some conflict resolver. The Earley suffix parser meets these defects by smoothly reverting to general suffix parsing, as described in Section 12.2.2.

Given the complexity of the Nederhof and Bertsch algorithm there is effectively only one linear-time LR suffix parser, the one discussed in Section 12.3.2. It is relatively easy to implement, makes good use of the LR(1) parse table, and is one of the few suffix parsing algorithms that have been used in real-world software. It can probably be relatively easily doctored to handle generalized LR(1) grammars; it then reverts to GLR suffix parsing.

12.4 Conclusion

Two efficient general suffix recognizers are available, one based on CYK and one based on Earley. Linear-time suffix recognition for LR(1) grammars can be achieved

by a simple adaptation of GLR parsing. A modified Earley parser or a tabular algorithm is required to obtain a linear-time LL(1) suffix recognizers. Algorithms for producing parse trees from the results of these recognizers are sorely underreported.

Next to substring parsing it is useful to consider subsequence parsing, where we have a sub*sequence* from a sentence in our the language and want to extract as much information as possible from it. Such subsequences result from transmissions with gaps in them, from partially damaged ancient texts, etc. Likewise it is interesting to look at superstring parsing and supersequence parsing, where the input contains one or more embedded sentences or fragments of a sentence in our language. Supersequences result among other things from noisy transmissions. Haines [387] has proved that both problems reduce to parsing a regular language, but we know of no research that explains how to apply Haines' results to parsing with a given CF grammar.

It should also be pointed out that both problems can, in principle, be solved through parsing by intersection (Chapter 13), but it is likely that more specific algorithms exist. Lavie and Tomita [173] adapt GLR parsing to do supersequence parsing, but the algorithm has exponential time requirement unless the search is limited by setting a heuristic parameter.

Problems

Problem 12.1: It is tempting to trivialize suffix parsing away by reversing both the grammar and the suffix. The suffix then becomes a prefix, and prefix parsing is well known. Comment.

Problem 12.2: In Section 12.1 we produce no "damaged" rule for $A \rightarrow \varepsilon$, but what happens if that is the only rule for A?

Problem 12.3: Extend any of the above suffix/substring parsers with a mechanism to create a completed parse tree.

Problem 12.4: Rather than remedying the lack of a start symbol in an Earley suffix parser (Section 12.2.2) by predicting all items, we could use the start symbol of the root set to start the sentential forms (Section 2.6). Elaborate.

Problem 12.5: The fact that we needed to modify the Earley-on-LL(1) parser to prove property *P1* and then could undo the modification without affecting the result shows that property *P1* is too strong. Devise the exact property needed.

Problem 12.6: In Section 12.3.3 we require each entry to contain a bounded number of lengths, but actually it is enough to require that the number of lengths in all entries together be bounded by a linear function in n. Can this additional leeway be exploited?

Problem 12.7: In the first paragraph of Section 12.3.3.1 we claim that starting an LL(1) parser for each entry in the recognition table is inefficient. Show that this is not true when memoization is used.

Problem 12.8: *Project:* Write a detailed algorithm to convert an LR(1) table into a restricted deterministic PDA (RDPDA) (page 420).

Problem 12.9: *Research problem:* It is conceivable that an algorithm for LR(1) suffix recognition can be fashioned along the lines of Section 12.3.3.1 as follows. Replace a left-recursive rule like $L \rightarrow L\alpha|\beta$ by the languages produced by α and β, L_α and L_β, and a regular expression $L = \beta\alpha^*$; if α or β are still left-recursive, repeat. Now all sublanguages are free of left recursion, and their non-terminals have at most a bounded number of lengths in each position. Make recognition tables for all these languages, and incorporate the regular expressions in the regular expression of the root matching automaton. Investigate this plan.

Problem 12.10: *Research problem:* There is considerable similarity but no obvious relationship between Nederhof and Bertsch's algorithm (Section 12.3.3.2) and Cook's linear-time simulation of a deterministic 2-way push-down automaton (2DPDA) [388]. Explore the similarity, for example by investigating the hypothesis "If a language is recognized by a DPDA, it is relatively simple to construct a 2DPDA that recognizes substrings in that language".

Problem 12.11: *Project*: Design a good algorithm for one of the three underresearched problems mentioned in the Conclusion to this chapter, using Haines's result [387] or otherwise.

13

Parsing as Intersection

In 1961 Bar-Hillel, Perles and Shamir [219] proved that "the intersection of a context-free language with a regular language is again a context-free language". On the face of it, this means that when we take the set of strings that constitute a given CF language and remove from it all strings that do not occur in a given FS language, we get a set of strings for which a CF grammar exists. Actually, it means quite a bit more.

It would be quite conceivable that the intersection of CF and FS were stronger than CF. Consider the two CF languages $L_1 = a^n b^n c^m$ and $L_2 = a^m b^n c^n$, whose CF grammars are in Figure 13.1. When we take a string that occurs in both languages, it

```
L1s  -->  A P
  A  -->  a A b | ε
  P  -->  c P | ε

L2s  -->  Q C
  C  -->  b C c | ε
  Q  -->  a Q | ε
```

Fig. 13.1. CF grammars for $a^n b^n c^m$ and $a^m b^n c^n$

will have the form $a^p a^q a^r$, where $p = q$ because of L_1 and $q = r$ because of L_2. So the intersection language consists of strings of the form $a^n b^n c^n$, and we know that that language is not context-free (Section 2.7.1). But Bar-Hillel et al.'s proof shows that that cannot happen with a CF language and a FS language. On the other hand one could well imagine that intersecting with a regular language would kill all CF power of description; again Bar-Hillel's proof shows that that is not the case.

Sometimes proofs of such theorems are non-constructive: one shows, for example, that if the theorem were not true one could solve a problem of which it has been proven that it is unsolvable, like full Type 0 parsing. Bar-Hillel et al.'s proof is much better than that: it is constructive, and demonstrates how to obtain the CF grammar of the intersection language, starting from the original CF grammar and the FS au-

tomaton that describes the regular language. And what is more, the process is quite simple and has polynomial (non-exponential) time requirements.

It was not realized until the 1990s that this has something to do with parsing. It is simple to see that an input string can be considered as a very simple, even linear, FSA with the positions between the tokens as the states. When, subsequently, parse forest grammars were recognized as a convenient way to represent the output of the parsing process (Billot and Lang [164]), the pieces began to fall together.

Parsing by intersection is based on three ideas:

- the input to the parsing process can be described by a finite-state automaton;
- the output of the parsing process can be described by a CF grammar;
- there is a practical process for obtaining the grammar for the intersection of a CF grammar with a finite-state automaton.

13.1 The Intersection Algorithm

Since even small examples will soon yield large data structures, we shall start with an almost microscopic grammar:

```
Ss  -->  a S
 S  -->  b
```

but we will also do examples with larger grammars further on. The input will be **ab**. As a finite-state automaton this defines 2 transitions and 3 states:

```
1 a 2 b 3
```

where 1 is the initial state and 3 the accepting state.

The intersection algorithm is unexpected but surprisingly simple. We start by creating a tentative non-terminal *A_n_m* for each non-terminal *A* in the original grammar G_{orig}, with the meaning that *A_n_m* produces everything that *A* produces and at the same time is recognized by the FS automaton between the states *n* and *m*. Next we derive rules from our grammar for all the *A_n_m*; how we do that is shown in the next section. This set of rules form the CF part of the intersection grammar; we will call it I_{rules}.

Two properties prevent I_{rules} from being a complete grammar: it has no start symbol and the treatment of terminal symbols is incomplete. To start with the latter, the rules in I_{rules} involve terminals of the form *t_p_q*, which describe those terminals *t* that provide a transition between states *p* and *q*. Now for each terminal *t* in the FSA that indeed provides a transition between states *p* and *q*, we construct a rule $t\_p\_q \rightarrow t$. And we declare all non-terminals *S_n_m* to be start symbols of the intersection grammar, where *S* is the start symbol of G_{orig}, *n* is the initial state of the FSA and *m* is an accepting state of the FSA. (Note that this may assign more than one start symbol to the intersection grammar.) Together with I_{rules} these constitute the rough form of the intersection grammar, I_{rough}.

And finally we clean I_{rough} using the algorithm from Section 2.9.5, which gives us the clean intersection grammar *I*. That's all. We will first see *that* it works and then *why* it works.

13.1.1 The Rule Sets I_{rules}, I_{rough}, and I

The tentative non-terminals from our demo grammar are

```
S_1_1, S_1_2, S_1_3
S_2_1, S_2_2, S_2_3
S_3_1, S_3_2, S_3_3
```

Suppose we want to create rules in I_{rules} for `S_1_3`. This non-terminal spans the FSA between the states 1 and 3, so its right-hand must span these two states too. There are two rules for `S`: `S→b` and `S→aS`. The first immediately yields a rule: `S_1_3→b_1_3`; but the second one involves an unknown state X: `S_1_3→a_1_`X `S_`X`_3`. Here the right-hand side spans 1…3 provided both occurrences of X are replaced by the same state, but that state is unknown. This is "solved" by brute force: we copy the rule for all states in the FSA:

```
S_1_3  →  a_1_1 S_1_3
S_1_3  →  a_1_2 S_2_3
S_1_3  →  a_1_3 S_3_3
```

We do this for all tentative non-terminals. This results in the rule set I_{rules}.

As usual, we have to pay some attention to ε-rules and ε-transitions. Our present example contains neither, but if the original CF grammar has a rule of the form $A \to \varepsilon$, a rule $A\_p\_p \to \varepsilon$ should be added to the intersection grammar I_{rules}, for each state p. Such rules represent the idea that an ε-producing A can occur in the parse tree without any consequence for the FSA. The intersection algorithm as published by Bar-Hillel et al. cannot handle ε-transitions in the FSA, but it is not very difficult to add this feature. See Problem 13.2.

Next we add the rules for the transitions in the FSA: `a_1_2→a` and `b_2_3→b`. To complete the intersection grammar, we appoint `S_1_3` as the start symbol. Since the initial state of the FSA is 1 and the accepting state is 3, `S_1_3` is going to produce whatever the FSA recognizes between its initial and accepting states.

All this amounts to 27 rules from `S→aS`, 9 rules from `S→b`, and 2 rules for the terminal symbols, for a total of 38 rules. They are collected in Figure 13.2, and together they form I_{rough}.

The last step consists of cleaning the intersection grammar. Removing the non-productive rules and unreachable non-terminals yields the clean intersection grammar I; it is shown in Figure 13.3. It is important to note that it is at the same time a parse-forest grammar, and, since the parsing is unambiguous, it is a parse-tree grammar. The corresponding parse tree is shown in Figure 13.4.

The parse forest above has been obtained in an almost shockingly simple way. The algorithm does not seem to need all the tricks and cleverness that we have met in previous algorithms. The parse forest grammar just somehow "develops" out of the original grammar, the way a photograph develops out of the latent image in photographic paper.

```
S_1_1 → a_1_1 S_1_1   S_2_1 → a_2_1 S_1_1   S_3_1 → a_3_1 S_1_1
S_1_1 → a_1_2 S_2_1   S_2_1 → a_2_2 S_2_1   S_3_1 → a_3_2 S_2_1
S_1_1 → a_1_3 S_3_1   S_2_1 → a_2_3 S_3_1   S_3_1 → a_3_3 S_3_1
S_1_2 → a_1_1 S_1_2   S_2_2 → a_2_1 S_1_2   S_3_2 → a_3_1 S_1_2
S_1_2 → a_1_2 S_2_2   S_2_2 → a_2_2 S_2_2   S_3_2 → a_3_2 S_2_2
S_1_2 → a_1_3 S_3_2   S_2_2 → a_2_3 S_3_2   S_3_2 → a_3_3 S_3_2
S_1_3S → a_1_1 S_1_3  S_2_3 → a_2_1 S_1_3   S_3_3 → a_3_1 S_1_3
S_1_3S → a_1_2 S_2_3  S_2_3 → a_2_2 S_2_3   S_3_3 → a_3_2 S_2_3
S_1_3S → a_1_3 S_3_3  S_2_3 → a_2_3 S_3_3   S_3_3 → a_3_3 S_3_3

S_1_1 → b_1_1         S_2_1 → b_2_1         S_3_1 → b_3_1
S_1_2 → b_1_2         S_2_2 → b_2_2         S_3_2 → b_3_2
S_1_3S → b_1_3        S_2_3 → b_2_3         S_3_3 → b_3_3

a_1_2 → a             b_2_3 → b
```

Fig. 13.2. Fully expanded intersection grammar

```
S_1_3S → a_1_2 S_2_3
S_2_3  → b_2_3
a_1_2  → a
b_2_3  → b
```

Fig. 13.3. Cleaned-up intersection grammar and parse forest grammar

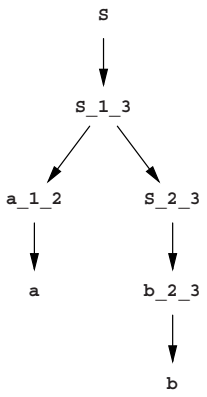


Fig. 13.4. Parse tree for the intersection grammar of Figure 13.3

13.1.2 The Languages of I_{rules}, I_{rough}, and I

To understand better how and why the algorithm works, we will consider the languages that are produced by the three successive versions of the intersection grammar, I_{rules}, I_{rough}, and I.

First we consider a grammar G_{rules} that has the set I_{rules} as its rules. As start symbols we accept all tentative non-terminals that derive from the original start symbol, `S_1_1` through `S_3_3`, and accept all FSA transitions `a_`*p*`_`*q* and `b_`*p*`_`*q* as its terminal symbols. With these provisions, G_{rules} produces all sequences of transitions `[a|b]_`*p*`_`*q* such that the sequence of `a`s and `b`s conform to the grammar G_{orig} and the *p*s and *q*s correctly link each transition to the next. A sample of the possible productions is

```
b_1_1
a_1_2 b_2_3
a_2_2 b_2_1
a_3_2 b_2_3
a_3_2 a_2_2 b_2_1
a_2_2 a_2_2 a_2_2 b_2_2
a_1_3 a_3_2 a_2_3 a_3_3 b_3_3
a_2_3 a_3_1 a_1_2 a_2_3 a_3_3 b_3_1
```

Restricting the start symbol to the only one that correctly specifies the initial and accepting states of the FSA, `S_1_3`, restricts the language produced to those sequences of transitions that start with `[a|b]_1_`*x* and end with `[a|b]_`*y*`_3`, for any state *x* and *y*. This restricted set contains sequences like

```
b_1_3
a_1_1 b_1_3
a_1_2 b_2_3
a_1_3 b_3_3
a_1_1 a_1_1 b_1_3
a_1_1 a_1_2 b_2_3
a_1_2 a_2_1 b_1_3
a_1_3 a_3_2 a_2_3 a_3_3 b_3_3
```

When we now restrict the transitions to those that are permitted by the FSA, `a_1_2` and `b_2_3`, only one sequence remains, `a_1_2 b_2_3`, which corresponds to our input string.

When we look carefully at the grammar cleaning process used to obtain I from I_{rough}, we see that it mirrors the above restriction process, except that it works on the grammar — which is finite — rather than on the produced language — which is infinite. Removing non-productive rules corresponds to removing non-permissible transitions, and removing unreachable symbols corresponds to removing start symbols that do not conform to the initial and accepting states of the FSA.

13.1.3 An Example: Parsing Arithmetic Expressions

We will now turn to a larger example, the parsing problem used in Section 4.1. We want to parse the sentence `(i+i)×i` with the grammar of Figure 4.1, here repeated in Figure 13.5. The input sentence corresponds to the FSA

```
Expr_S  →  Expr + Term | Term
 Term   →  Term × Factor | Factor
Factor  →  ( Expr ) | i
```

Fig. 13.5. A grammar describing simple arithmetic expressions

```
1 ( 2 i 3 + 4 i 5 ) 6 × 7 i 8
```

which is a convenient short-hand for the seven rules

```
(_1_2  →  (
   ...
i_7_8  →  i
```

Since the rough intersection grammar I_{rough} has 12487 rules, it is not possible to demonstrate the complete process here, and the reader will have to believe us when we say that the resulting cleaned grammar I is shown in Figure 13.6. When we

```
 Expr_2_5  →  Expr_2_3 +_3_4 Term_4_5
Expr_1_8_S →  Term_1_8
 Expr_2_3  →  Term_2_3
 Term_1_8  →  Term_1_6 ×_6_7 Factor_7_8
 Term_1_6  →  Factor_1_6
 Term_2_3  →  Factor_2_3
 Term_4_5  →  Factor_4_5
Factor_1_6 →  (_1_2 Expr_2_5 )_5_6
Factor_2_3 →  i_2_3
Factor_4_5 →  i_4_5
Factor_7_8 →  i_7_8
```

Fig. 13.6. Intersection grammar for the sentence `(i+i)×i`

compare this grammar to the parse forest grammar obtained from the Unger parser in Figure 4.5, we see that the two are almost identical, and that the differences are minor. In Unger parsing a grammar symbol is marked with the start position and length of the segment it spans, whereas in intersection parsing it is marked with the states of the FSA. A slightly more characteristic difference is that the rules in the Unger parse forest grammar are produced in top-down order, whereas those in the intersection grammar appear in a seemingly arbitrary order, determined by the order of the rules in the original grammar and the details of the cleaning algorithm.

13.2 The Parsing of FSAs

Until now we have only used linear FSAs, thereby restricting ourselves to tasks that could be performed equally well or even better by traditional parsing. We will now consider intersection with more general FSAs: FSAs for arithmetic expressions with one or more unknown tokens in them, FSAs for substrings of arithmetic expressions, and the FSA for `?`*, where `?` is any token in the grammar.

13.2.1 Unknown Tokens

We start with the regular expression `(i?i)×i`; it represents an input sentence of which the third token is unknown. In terms of the FSA this means that where we had the rule `+_3_4 → +` in Section 13.1.3, we now have the rules

```
i_3_4  → i
+_3_4  → +
×_3_4  → ×
(_3_4  → (
)_3_4  → )
```

Two new rules appear in the parse forest grammar, compared to Figure 13.6:

```
Expr_2_5  → Term_2_5
Term_2_5  → Term_2_3 ×_3_4 Factor_4_5
```

These supply a new alternative for `Expr_2_5`, and show that the algorithm has recognized that `(i×i)×i` is a terminal production of `Expr` too, but since the × is in a different place in the grammar, an additional step through `Term_2_5` is required. Note that we now have a parse *forest* since there are two alternatives for `Expr_2_5`.

Something completely different happens when we try the regular expression `(i??i)×i`, since we end up with an empty grammar: the cleaning process removes even the start symbol. This is correct: no terminal production of `Expr` matches `(i??i)×i`.

Again a different effect is obtained from the regular expression `(i???i)×i`. A fairly long (30 rules), fairly uninformative grammar results, which contains several non-terminals with multiple alternatives, and which produces the following six sentences:

```
( i × i × i ) × i
( i ) × ( i ) × i
( i + i × i ) × i
( i × i + i ) × i
( i ) + ( i ) × i
( i + i + i ) × i
```

13.2.2 Substring Parsing by Intersection

Intersection parsing can also be used to do substring parsing. For example, to parse the substring `+i)`, the FSA corresponding to `?`*`+i)?`* is offered to the intersection process. This FSA is represented by the rules

```
?_1_1  →  ?
+_1_2  →  +
i_2_3  →  i
)_3_4  →  )
?_4_4  →  ?
```

where the question mark is an abbreviation for all terminals in the grammar.

The intersection process results in the parse forest grammar of Figure 13.7. This

```
  Expr_1_1  →  Expr_1_1 +_1_1 Term_1_1 | Term_1_1
  Expr_1_3  →  Expr_1_1 +_1_2 Term_2_3
 Expr_1_4s  →  Expr_1_1 +_1_1 Term_1_4 |
               Expr_1_4 +_4_4 Term_4_4 | Term_1_4
  Expr_4_4  →  Expr_4_4 +_4_4 Term_4_4 | Term_4_4
  Term_1_1  →  Term_1_1 ×_1_1 Factor_1_1 | Factor_1_1
  Term_1_4  →  Term_1_1 ×_1_1 Factor_1_4 |
               Term_1_4 ×_4_4 Factor_4_4 | Factor_1_4
  Term_4_4  →  Term_4_4 ×_4_4 Factor_4_4 | Factor_4_4
  Term_2_3  →  Factor_2_3
Factor_1_1  →  (_1_1 Expr_1_1 )_1_1 | i_1_1
Factor_1_4  →  (_1_1 Expr_1_3 )_3_4 | (_1_1 Expr_1_4 )_4_4
Factor_4_4  →  (_4_4 Expr_4_4 )_4_4 | i_4_4
Factor_2_3  →  i_2_3
```

Fig. 13.7. Parse forest grammar for the substring `+i)`

```
 Expr_1_4s  →  Expr_1_4 + Term | Expr + Term_1_4 | Term_1_4
  Term_1_4  →  Term_1_4 × Factor | Term × Factor_1_4 |
               Factor_1_4
Factor_1_4  →  ( Expr_1_3 )_3_4 | ( Expr_1_4 )
  Expr_1_3  →  Expr +_1_2 Term_2_3
  Term_2_3  →  Factor_2_3
Factor_2_3  →  i_2_3
      Expr  →  Expr + Term | Term
      Term  →  Term × Factor | Factor
    Factor  →  ( Expr ) | i
```

Fig. 13.8. A manually simplified version of the grammar of Figure 13.7

grammar produces all sentences produced by the original grammar in Figure 13.5 that contain a substring `+i)`. With its 12 rules it is remarkably simple, certainly for a grammar that was produced automatically, for what at first sight would not seem to be a simple problem. It is even reasonably easy to see what it does: the nonterminals marked `_1_1` and `_4_4` produce expressions to the left and the right of the substring, and only those with "mixed" markings are concerned with fitting them around the substring.

This is even more visible in a manually simplified version of the same grammar, shown in Figure 13.8. The non-terminals marked **_1_1** and **_4_4** have been simplified into one copy of the original grammar for **Expr**; terminal productions of this grammar are not guaranteed to contain the substring; note that this **Expr** is not the start symbol. The non-terminals marked **_1_4**, on the other hand, are obliged to contain the substring; **Expr_1_4** and **Term_1_4** parcel out this obligation to their left child, their right child or their only child. **Factor_1_4** is the first rule that can actually form part of the substring by producing the **)_3_4**. The remaining obligation to span the states 1 and 3 is passed to **Expr_1_3**, from where it trickles down. Note that none of the above considerations are present in the intersection algorithm; the algorithm just expands and cleans.

An attempt to draw the corresponding parse forest is shown in Figure 13.9. Several simplifications have been applied: the unmarked **Expr**, **Term**, and **Factor**

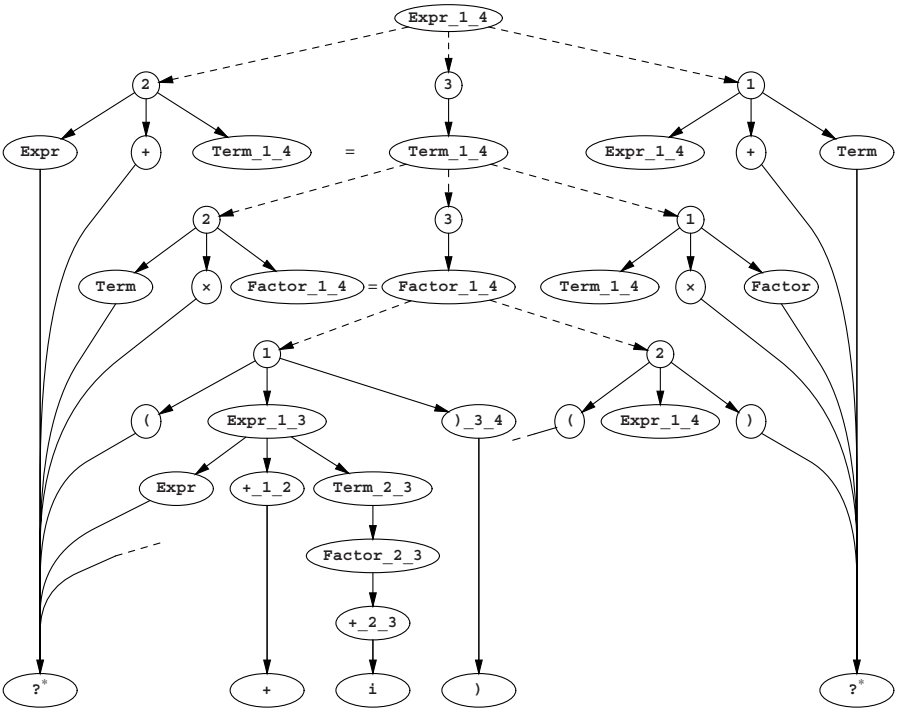


Fig. 13.9. Simplified parse forest for the grammar of Figure 13.8

have not been expanded; (marked) non-terminals that occur more than once have been expanded only once; and OR-nodes are not shown when there are no alternatives. The numbers in the other OR-nodes are the numbers of the alternatives in the rule for the parent in Figure 13.8.

Another interesting regular expression is `?(?`*, which describes all sentences that have an open parenthesis as their second symbol. The intersection grammar is again simplicity itself: Figure 13.10. It is quite remarkable that the only symbol with marking `_1_2` is an open parenthesis: the intersection process has "discovered" that if the second symbol of an expression is an open parenthesis, the first symbol must be an open parenthesis too! Manual simplification yields the grammar of Figure 13.11; we have left out the original grammar for **`Expr`**.

We started this chapter with Bar-Hillel's theorem that "the intersection of a context-free language with a regular language is again a context-free language", but until now we have created and discussed just the CF grammars of those languages. The grammars in Figures 13.10 and 13.11 give us the opportunity to view them as grammars which produce "new" languages, rather than as parse forest grammars which represent parsings. The difference between the two is a matter of degree.

```
Expr_1_3_S  →  Expr_1_3 +_3_3 Term_3_3 | Term_1_3
 Expr_2_3   →  Expr_2_3 +_3_3 Term_3_3 | Term_2_3
 Expr_3_3   →  Expr_3_3 +_3_3 Term_3_3 | Term_3_3
 Term_1_3   →  Term_1_3 ×_3_3 Factor_3_3 | Factor_1_3
 Term_2_3   →  Term_2_3 ×_3_3 Factor_3_3 | Factor_2_3
 Term_3_3   →  Term_3_3 ×_3_3 Factor_3_3 | Factor_3_3
Factor_1_3  →  (_1_2 Expr_2_3 )_3_3
Factor_2_3  →  (_2_3 Expr_3_3 )_3_3
Factor_3_3  →  (_3_3 Expr_3_3 )_3_3 | i_3_3
```

Fig. 13.10. A grammar for all arithmetic expressions that have a `(` for their second symbol

```
Expr_1_3_S  →  Expr_1_3 + Term | Term_1_3
 Expr_2_3   →  Expr_2_3 + Term | Term_2_3
 Term_1_3   →  Term_1_3 × Factor | Factor_1_3
 Term_2_3   →  Term_2_3 × Factor | Factor_2_3
Factor_1_3  →  (_1_2 Expr_2_3 )
Factor_2_3  →  (_2_3 Expr )
```

Fig. 13.11. A manually simplified version of the essential part of the grammar in Figure 13.10

At one extreme, a parse tree can be viewed as a grammar that produces exactly one sentence; at the other, a grammar as in Figure 13.5 can be viewed as a parse forest for all arithmetic expressions. We will see more evidence for this view in the next paragraph and in Section 13.2.3.

It is also interesting to see the result of intersecting with the FSA that accepts any sequence, the FSA for `?`*. This FSA has one state, 1, and all tokens have transitions from 1 to 1. The intersection is so simple in this case that it can be performed by hand. The outcome is in Figure 13.12, and, except for the markings `_1_1`, it is identical to

```
Expr_1_1S  ->  Expr_1_1 +_1_1 Term_1_1 | Term_1_1
 Term_1_1  ->  Term_1_1 x_1_1 Factor_1_1 | Factor_1_1
Factor_1_1 ->  (_1_1 Expr_1_1 )_1_1 | i_1_1
```

Fig. 13.12. The grammar from Figure 13.5 filtered through an FSA that accepts any sequence

the original grammar. This is of course as it should be, but it is still satisfying to see that it is.

13.2.3 Filtering

Since both the input and the output of the intersection process are grammars, intersection can be used as a filter: the result of intersection with an FSA can again be intersected with another FSA. We shall use this technique to construct a grammar for the language $\mathtt{a}^p\mathtt{b}^q\mathtt{c}^r$, $p,q,r \geq 1$, by filtering a grammar for arbitrary sequences of **a**s, **b**s, and **c**s through an FSA which disallows **b**s before **a**s and then through an FSA which disallows **c**s before **b**s.

We start from the obvious grammar for **[abc]***

```
Ss  ->  a S
 S  ->  b S
 S  ->  c S
 S  ->
```

and filter it through **[ac]*****a[bc]***. This regular expression describes a sequence of **a**s and **c**s with at least one **a** followed by a sequence of **b**s and **c**s; that is, it forces all **a**s to come before all **b**s, without affecting the **c**s. The result of the intersection is

```
S_1_2S  ->  a S_1_2
 S_1_2  ->  a S_2_2
 S_2_2  ->  b S_2_2
 S_1_2  ->  c S_1_2
 S_2_2  ->  c S_2_2
 S_2_2  ->
```

where we have replaced the marked terminals **a_1_1** etc. by their unmarked counterparts, since we are interested in the resulting grammar rather than in the parsing of the FSA for **[ac]*****a[bc]***.

Filtering this grammar through the regular expression **[ab]*****b[ac]***, which requires at least one **b** and forces all **b**s to come before all **c**s, yields the grammar

```
S_1_2_1_2S  ->  a S_1_2_1_2
 S_1_2_1_2  ->  a S_2_2_1_2
 S_2_2_1_2  ->  b S_2_2_1_2
 S_2_2_1_2  ->  b S_2_2_2_2
 S_2_2_2_2  ->  c S_2_2_2_2
 S_2_2_2_2  ->
```

Now the only thing left is to require at least one **c** and we do this by filtering through the regular expression **[abc]*****c**. The resulting grammar is

```
S_1_2_1_2_1_2  →  a S_1_2_1_2_1_2
S_1_2_1_2_1_2  →  a S_2_2_1_2_1_2
S_2_2_1_2_1_2  →  b S_2_2_1_2_1_2
S_2_2_1_2_1_2  →  b S_2_2_2_2_1_2
S_2_2_2_2_1_2  →  c S_2_2_2_2_1_2
S_2_2_2_2_1_2  →  c S_2_2_2_2_2_2
S_2_2_2_2_2_2  →
```

We see again that the grammar does not become unduly large, but it does look forbidding, until we realize that it contains only four non-terminals, `S_1_2_1_2_1_2`, `S_2_2_1_2_1_2`, `S_2_2_2_2_1_2`, and `S_2_2_2_2_2_2`. If we rename them to `S`, `T`, `U`, and `V`, we obtain the grammar

```
S  →  a S
S  →  a T
T  →  b T
T  →  b U
U  →  c U
U  →  c V
V  →
```

which is one of the simplest grammars for the language $\mathtt{a}^p\mathtt{b}^q\mathtt{c}^r$, $p,q,r \geq 1$.

Note that this grammar was derived without applying any human intelligence. The final rewrite was only cosmetic and served just to make the result more readable.

13.3 Time and Space Requirements

The reader will already have noticed that the grammars resulting from intersection parsing are of modest sizes; twenty to thirty rules for an input of say ten tokens. This is not really surprising, since the grammars represent parse trees, which for unambiguous input have sizes ranging from linear ($O(n)$) to $O(n\ ln\ n)$ to $O(n^2)$ for the worst case.

But appearances are deceptive in this case, since the intersection process is one of expansion and clean-up, and the intermediate data structures are much larger. We have seen that the fully expanded grammar for the parsing of `(i+i)×i` with the grammar of Figure 13.5 contains 12487 rules. This number is easily verified: There are 8 states. Each rule with a right-hand side of length 1 $A \rightarrow B$ expands into 8×8 rules *A_n_m* → *B_n_m* , for all 8 values of n and m; and there are 3 of these. Each rule with a right-hand side of length 3 $A \rightarrow BCD$ expands into $8 \times 8 \times 8 \times 8$ rules *A_n_p* → *B_n_m C_m_o D_o_p*, for all 8 values of n, m, o and p; and there are again 3 of these. This gives $(3 \times 8^2 = 3 \times 64 = 192) + (3 \times 8^4 = 3 \times 4096 = 12288) = 12480$ rules, plus 7 for the terminals makes 12487. More in general, a rule with a right-hand side of length k expands into n^{k+1} rules, where n is the number of states in the FSA. For a linear FSA with n states, representing a string of length $n-1$, and a grammar in CNF, this reduces to n^3, exactly the space requirement of the traditional general CF parsing algorithms.

13.4 Reducing the Intermediate Size: Earley's Algorithm on FSAs

We have seen that the size of the intermediate grammar may be a problem. One immediate idea to address this problem is to suppress the generation of all rules that contain non-existent marked tokens: when we are about to generate a rule like **Expr_1_1→Expr_1_1 +_1_1 Term_1_1** and know there is no **+_1_1**, we refrain from doing so. This optimization reduces the number of rules in I_{rough} from 12487 to 260 (of which 11 remain after clean-up).

Another approach is trying to be as frugal as we can in creating rules, as follows. To keep the size of the example manageable, we consider the grammar

$$\begin{array}{lll} \mathtt{S}_\mathtt{S} & \rightarrow & \mathtt{a\ S\ b} \\ \mathtt{S} & \rightarrow & \varepsilon \end{array}$$

which produces the language $\mathtt{a}^n\mathtt{b}^n$, and which we are going to intersect with the FSA for $\mathtt{a}^*\mathtt{bb}$, represented by the grammar rules

$$\begin{array}{lll} \mathtt{a\_1\_1} & \rightarrow & \mathtt{a} \\ \mathtt{b\_1\_2} & \rightarrow & \mathtt{b} \\ \mathtt{b\_2\_3} & \rightarrow & \mathtt{b} \end{array}$$

Here 1 is the initial state and 3 is the accepting state.

We certainly need rules that start with the start symbol of the grammar, in the initial state of the FSA: **S_1_*X*→a_1_*Y* S b** and **S_1_1→ε_1_1**. Both start in FSA state 1. The first rule ends in an as yet unknown state *X*. Its right-hand side starts with an **a** starting in FSA state 1 and ending in another as yet unknown state *Y*. There are also other unknown states connecting **a** to **S** and **S** to **b**, so the full form of the rule is **S_1_*X* → a_1_*Y* S_*Y*_*Z* b_*Z*_*X***; note that the end state of **b** is *X*, the end state of the left-hand side. We do not know yet what the states are, but we know that they are equal. The second rule also starts in state 1, which makes the ε start in state 1, which of course ends in state 1, which in its turn makes the **S** end in state 1. For convenience the rules generated by the algorithm are brought together in Figure 13.13. It starts with the rules for the terminal symbols since these will be

1.	**a_1_1 → a**		from the FSA
2.	**b_1_2 → b**		from the FSA
3.	**b_2_3 → b**		from the FSA
4.	**S_1_*X* → a_1_*Y* S_*Y*_*Z* b_*Z*_*X***		from the start symbol and from 6
5.	**S_1_1 → ε_1_1**		from the start symbol and from 6
6.	**S_1_*X* → a_1_1 S_1_*Z* b_*Z*_*X***		from 4 and 1
7.	**S_1_*X* → a_1_1 S_1_1 b_1_*X***		from 6 and 5
8.	**S_1_2 → a_1_1 S_1_1 b_1_2**		from 7 and 2
9.	**S_1_*X* → a_1_1 S_1_2 b_2_*X***		from 6 and 8
10.	**S_1_3 → a_1_1 S_1_2 b_2_3**		from 9 and 3
11.	**S_1_*X* → a_1_1 S_1_3 b_3_*X***		from 6 and 10; dead end

Fig. 13.13. The rules generated by the Earley-like intersection algorithm

needed in any case.

To complete the intersection grammar we need to find instantiations of the uninstantiated variables in them. The first rule with an uninstantiated variable is rule 4. We can instantiate X only once we have identified the **b** at the end, so we try to find a value for the Y. The only way to do this is to go to the rules that have already been generated and find there a rule with a left-hand side **a_1_**k where k is instantiated; there is only one such rule, rule 1. (We are using upper case letters (X, Y, etc.) for uninstantiated variables and lower case letters (k, p, etc.) for variables that have values.) Instantiating the Y to 1 yields rule 6, where the next position in need of instantiation has moved to the **S** in the right-hand side.

Now there are two ways to use rule 6. One is to consult again the already generated rules and find a rule with a left-hand side **S_1_**k where k is instantiated. There is one such rule in Figure 13.13, rule 5; combining it with rules 6 yields rule 7, in which only X is still unknown. But since **S** is a non-terminal, there is another way: to consult the original grammar and instantiate new rules for **S_1_**Z. The two rules for **S** in the grammar yield rules that are identical to rules 4 and 5, so this step creates no new rules.

Rule 7 requires a **b_1_**k which is provided by rule 2. Here X finally receives a value, 2, from the **b_1_2**. This means that a new **S_1_**k with k instantiated has been created, **S_1_2**. This allows a new possibility to derive a rule from rule 6: rule 9. The missing X in **b_2_**X is provided by rule 3, instantiating X to 3; this results in rule 10. So again a new **S_1_**k has been created, **S_1_3**, which again yields a new rule from rule 6: rule 11. But since there is no rule that supplies a **b_3_**k, this is a dead end, and since no more new rules can be derived from the existing ones and the grammar, the algorithm stops here.

As a final step we clean the grammar by removing all rules that contain uninstantiated variables, mark **S_1_3** as the start symbol and thus obtain the intersection grammar

```
a_1_1   →  a
b_1_2   →  b
b_2_3   →  b
S_1_1   →  ε_1_1
S_1_2   →  a_1_1 S_1_1 b_1_2
S_1_3_S →  a_1_1 S_1_2 b_2_3
```

This grammar indeed produces **aabb** with the proper parse tree.

The reader will have noticed (if it were only from the title of this section) that the algorithm we have shown above is a variant of the Earley algorithm described in Section 7.2. The relationship can be summed up as follows.

- A tentative rule of the form $A\_p\_X \rightarrow B\_p\_ \cdots C\_q\_Y \cdots$ corresponds to an item $A \rightarrow B \cdots C\bullet \cdots @p$ in $itemset_q$ in the traditional Earley parser. The correspondence is not exact, though. If the Earley parser has an item $A \rightarrow BC\bullet \cdots @p$ in $itemset_q$ and there are two ways in which BC can produce the input segment from p to q with different lengths for the Bs and Cs, the intersection Earley parser has two tentative rules, $A\_p\_X \rightarrow B\_p\_i\ C\_i\_q \cdots$ and $A\_p\_X \rightarrow B\_p\_j\ C\_j\_q \cdots$,

with $i \neq j$. This is because the intersection Earley parser constructs the parse forest on the fly and does not need a second scan (Section 7.2.1.2) to construct the parse forest.

- Creating new rules from a tentative rule $A\_p\_X \rightarrow B\_p\_ \cdots V\_q\_Y \cdots$ by finding symbols of the form V_q_r in G_{rough}, where V is a terminal or a non-terminal, corresponds to the actions of the Scanner.
- Creating new rules from a tentative rule $A\_p\_X \rightarrow B\_p\_ \cdots C\_q\_Y \cdots$ by finding rules of the form $C \rightarrow \cdots$ in G_{orig} corresponds to the actions of the Predictor.
- Completing a tentative rule of the form $A\_p\_X \rightarrow B\_p\_ \cdots E\_q\_r$ to $A\_p\_r \rightarrow B\_p\_ \cdots E\_q\_r$ and then offering it to the Scanner corresponds to the action of the Completer.

This discussion of the intersection Earley parser is based on Section 5 of Albro's paper [222], which has the intersection and the Earley part, but not the parse forest grammar part.

One wonders if the LL(1) and LR(1) methods can also be extended to parse finite-state automata, but to our knowledge no such research has been reported. See Problem 13.7 for some thoughts.

13.5 Error Handling Using Intersection Parsing

We have seen (page 431) that if we intersect a grammar with an incorrect input sequence (or actually with its FSA) an empty grammar results. From the viewpoint of the algorithm nothing is wrong. The algorithm has dutifully computed the intersection and the intersection happened to be empty.

We have also seen that intersection parsing can be used to do substring parsing, and the idea suggests itself to use this capability to do error handling. This does not work either since unless we first identify a correct substring in the incorrect input (and how could we do that) the intersection algorithm will again produce an empty result.

When we follow the algorithm in what it does on its way to rejecting all rules of the intersection grammar, we see that it collects valuable information bottom-up, but that all that information is deleted because it is not reachable from the start symbol. This is easy to understand: error information can only be collected bottom-up, since the bottom-up process collects correct building blocks; top-down the answer to any incorrect situation is always "No". So the idea suggests itself to stop the cleaning phase *before* unreachable rules are removed, and extract the error information from that stage. We will call the grammar at that stage G_{BU} for "Bottom-Up".

As an example, we will try to parse the erroneous string `(i+i)+×i` using the grammar from Figure 13.5; this is the same string as we used in Section 13.1.3 with a spurious `+` inserted. The FSA corresponding to the input string is

```
1 ( 2 i 3 + 4 i 5 ) 6 + 7 × 8 i 9
```

and the resulting G_{BU} is shown in Figure 13.14. Two useful bits of information can

```
  Expr_2_5  →  Expr_2_3 +_3_4 Term_4_5
  Expr_1_6  →  Term_1_6
  Expr_2_3  →  Term_2_3
  Expr_4_5  →  Term_4_5
  Expr_8_9  →  Term_8_9
  Term_1_6  →  Factor_1_6
  Term_2_3  →  Factor_2_3
  Term_4_5  →  Factor_4_5
  Term_8_9  →  Factor_8_9
Factor_1_6  →  (_1_2 Expr_2_5 )_5_6
Factor_2_3  →  i_2_3
Factor_4_5  →  i_4_5
Factor_8_9  →  i_8_9
```

Fig. 13.14. Bottom-up part of the intersection grammar

be derived immediately from this grammar. The first is that the grammar contains all terminals except `+_6_7` and `×_7_8`, so these tokens are implicated in the error. The second is that **`Expr_1_6`**, spanning 5 tokens, is the longest identified node in the parsing.

The non-terminal **`Expr_1_6`** describes the largest correct subtree over the input; it explains the segment `(i+i)`. When we isolate it from the grammar, we can find the next largest, etc. The next one we find is headed by **`Expr_8_9`**; it explains the final `i`. No further subtrees can be found; the remaining non-terminal **`Expr_4_5`** does not lead anywhere. The subtrees are shown in Figure 13.15. It is not immedi-

```
                    Expr_1_6  →  Term_1_6
                    Term_1_6  →  Factor_1_6
                  Factor_1_6  →  (_1_2 Expr_2_5 )_5_6
                    Expr_2_5  →  Expr_2_3 +_3_4 Term_4_5
Largest subtree:    Expr_2_3  →  Term_2_3
                    Term_2_3  →  Factor_2_3
                  Factor_2_3  →  i_2_3
                    Term_4_5  →  Factor_4_5
                  Factor_4_5  →  i_4_5

                    Expr_8_9  →  Term_8_9
One but largest subtree:
                    Term_8_9  →  Factor_8_9
                  Factor_8_9  →  i_8_9
```

Fig. 13.15. Two subtrees identified by the bottom-up part of the intersection process

ately clear how this information can be converted into helpful error messages; one possibility is telling the user that the input can be reduced to **`Expr + × Expr`** but not further.

Two things can be noticed here. The first is that a directional parser would have accepted the last `+` in `(i+i)+×i`, and declared the `×` to be the culprit. But from a non-directional point of view — and intersection parsing is a non-directional method — there is nothing that says that the `×` is more in error than the `+`. See also Problem 13.8.

The second is that the above technique allows us to identify subsections of FSAs that match subsections of CF grammars. It is not clear what that means, or how or where that can be used.

13.6 Conclusion

Parsing as intersection immediately provides polynomial-time algorithms for tasks that would seem problematic otherwise, including substring parsing, managing ambiguity, and large-scale error recovery, all without imposing restrictions on the CF grammar used in the parsing. Although its basic component is old, it is a relatively new and little-studied subject; (Web)Section 18.2.4 holds only 6 references.

In spite of its title, two subjects have been introduced in this chapter: intersection parsing and parsing of finite-state automata. The combination is fortunate, but each is also valuable on its own accord. Intersection parsing allows new insights in and representations of ambiguous parsings, and may play a role in the unification of the present plethora of parsing algorithms. The ease with which the Earley algorithm was derived (Section 13.4) bodes well in this respect. The parsing of FSAs allows the analysis of damaged or incorrect input in a natural way; extension of LL, LR and generalized LR techniques to FSAs would be very useful. Not enough research has been done to fully appreciate the possibilities of Bar-Hillel's long-neglected algorithm.

The results in this chapter have been obtained in an unexpected and surprising way. Especially the determination by the algorithm that the first token in a simple arithmetic expression of which the second one is an open parenthesis (Section 13.2.2) must also be an open parenthesis is astonishing. One wonders where the cleverness and the intelligence is that seemed so necessary in all our previous parsing algorithms. Part of the answer seems to lie in the use of names of non-terminals as multi-way pointers, but it may be too early to find a satisfactory answer to this question.

Problems

Problem 13.1: The use of logic variables in Section 13.1.1 suggests that the intersection of FSAs and CF grammars can be implemented conveniently in Prolog, which has logic variables as a built-in feature. Explore such an implementation.

Problem 13.2: Extend Bar-Hillel's intersection algorithm so it can handle ε-transitions in the FSA.

Problem 13.3: *Project* To be honest, the three regular expressions used in Section 13.2.3 to create a CF grammar for the language $\mathtt{a}^p\mathtt{b}^q\mathtt{c}^r$ were chosen carefully to produce a nice grammar. Unfortunately, most other regular expressions one could use for this purpose, for example `[ac]`*`[bc]`*, produce less attractive results. Investigate this phenomenon.

Problem 13.4: Turn the sketch of the intersection Earley parser in Section 13.4 into a complete algorithm.

Problem 13.5: We could refuse to produce rule 11 in Figure 13.13 on the grounds that there is no `b_3_`k. This corresponds to prediction look-ahead in the Earley parser, as described in Section 7.2.4.1. Incorporate this optimization in the result of Problem 13.4.

Problem 13.6: *Project:* The Earley intersection parser creates (as frugally as possible) the rules from the top down, but one could also create them bottom-up. Start with all terminals of the form t_p_q, find all rules that contain t, and instantiate as many positions in the non-terminals as possible. On the basis of these half-constructed rules, create more rules. etc. Turn this into an algorithm.

Problem 13.7: *Research Project:* (*a*) How can the LL(1) parsing method be adapted to parse finite-state automata rather than strings? Hint: The big question is of course what to do about the prediction stack. It seems possible to construct, for each state p of the FSA F to be parsed, an FSA G_p which describes all the prediction stacks that any string produced by F could encounter in state p, either by specializing the general FSA for the prediction stack (Section 5.1.1) or by propagation and transitive closure. (*b*) Same question for LR(1). (*c*) Same question for Generalized LR.

Problem 13.8: Error detection traditionally requires the determination of the longest grammatically acceptable prefix of the input string. Consider an algorithm that determines that prefix by binary search, using the intersection algorithm as a test of acceptability. What is the complexity of this algorithm? Once the prefix has been removed, can a similar algorithm be used to determine further correct substrings?

14

Parallel Parsing

There are two main reasons for doing parallel programming: the problem has inherent parallelism, in which case the parallel programming comes naturally; and the problem is inherently sequential but we need the speed-up promised by parallel processing, in which case the parallel programming is often a struggle.

Parsing does not fall in either of these categories. It has no obvious or inherent parallelism and on present-day machines and using state-of-the-art techniques it is already very fast. A 250-line module in a computer program and a 25-word sentence in a natural language can both be parsed in a fraction of a second. In parallel parsing processing is performed on multiple processors, which either have a shared memory or are interconnected by means of a (high speed) network. Given the communication overhead in the most common parallel systems, little speed-up can be expected for the average parsing task.

14.1 The Reasons for Parallel Parsing

From a practical point of view, parallel parsing is interesting only for problems big enough to require considerably more time than a fraction of a second on a single processor. There are three ways in which a parsing problem can be this big: the input is very long (millions of tokens); the grammar is very large (millions of rules); or there are millions of inputs to be parsed. The last problem can be solved trivially by distributing the inputs over multiple processors, where each processor processes a different input and runs an ordinary, sequential, parser.

Examples of very long inputs requiring parsing are hard to find. All very long parsable sequences occurring in practice are likely to be regular: generating very long CF sequences would require a place to store the nesting information during sentence generation. The boundaries are not very clear, though, since a novel can be considered a very long parsable sequence of words. Upon a closer look, this sequence, however, is either regular — just a list of chapters which are lists of paragraphs which are lists of sentences, which in themselves are CF — or it is context-sensitive — a coherent sequence of words, punctuation, etc., with strong context dependencies, for

example keeping track of the color of the heroine's eyes. This suggests that parallel context-sensitive parsing might be useful, but no research on the subject is known.

The situation is different for parsing with very large grammars. These are found most often in linguistics. They are especially bothersome there since most linguistic applications require general CF parsing techniques, the speed of which depends on the grammar size. This dependency is quadratic ($O(|G|^2)$, where $|G|$ is the size of the grammar), or, for more advanced algorithms, linear ($O(|G|)$). General CF parsing has a practical upper bound of $O(n^3)$, where n is the length of the sentence, but if the grammar is very large and the sentence to be parsed is short, grammar size may be much more significant then input length. Little explicit research on parsing with very large grammars is known, but many parallel parsing techniques can be exploited to allow very large grammars, as we shall see.

This brings us to two further reasons to study parallelism: scientific curiosity and theoretical investigations. It is well known that trying to parallelize an otherwise sequential program often leads to deeper insight, clever techniques, and sometimes to improved infrastructure. The theoretical investigations are concerned with the inherent complexity of parsing: Can parsing be done in linear time ($O(n)$)? (Yes, easily, using $O(n^3)$ processors; see Section 14.4.2.1.) Can parsing be done in logarithmic time ($O(\ln n)$)? (Yes, with difficulty, using $O(n^6)$ processors; see Sections 14.4.3.1 and 14.4.3.3.) Can parsing be done in double-logarithmic time ($O(\ln \ln n)$)? (We don't know.)

It is because of these considerations and questions that researchers have given parallel parsing a lot of attention, so much so that it is impossible to describe all this research in a single chapter. Rather, we will describe some of the directions this research has taken.

Three main methods of parallelizing the parsing process have been developed: multiple serial parsers, process-configuration parsers, and connectionist parsers. These correspond closely to three main streams in the use of multiple processors: parallel programming, in which most of the processors perform the same program with different data sets; distributed programming, in which most of the processors perform autonomous cooperating programs; and hardware parallelism, in which large numbers of processors consisting of special hardware perform simple actions.

In this chapter we will discuss an example of each of the parsing methods.

14.2 Multiple Serial Parsers

A multiple serial parsers is a parallel parser in which each processor runs a sequential parser on a part of the input, which is split up and divided among the processors. A typical example of a multiple serial parser is Fischer's algorithm [223]. Fischer developed a mechanism that can be applied to several sequential parsing techniques. We will discuss the LR(0) version here.

Each Fischer parser has a number of incomplete LR(0) stacks, each incomplete stack being the head (top segment) of a possible actual stack. N such parsers are started at N essentially arbitrary points in the input sequence (which must be known

in its entirety); one parser starts at the beginning. The latter starts with one stack head, containing the initial state, so it certainly starts with a correct and complete LR(0) stack. The stack head sets of the other parsers are determined by the first token they look at. If a certain parser looks at a token *a*, it gets one stack head for each state in the LR(0) table that allows a shift on *a*, and that stack head contains that state. Each parser proceeds as follows: It considers each active state in the stack head set; some will indicate a shift, some a reduce and some will indicate an error. The stack heads indicating an error are discarded. Now, five cases are distinguished:

- All active states (the states on top of the stack heads) indicate a shift: the action is done for each stack head and the parser proceeds.
- All active states indicate a reduction for which the corresponding stack head is deep enough: the action is done for each stack head and the parser proceeds. Note that the stack head of the first parser will always be deep enough.
- All active states indicate a reduction, but none of the stack heads are deep enough to allow the reduction. In this case, the parser suspends itself, it has to wait for its left neighbor.
- There are no stack heads left; all stack heads were discarded because they indicated an error. In this case, there actually was an error.
- If none of the above applies, the parser splits the stack head set up in such a way that one of the above cases applies to each part. Next, a number of further parsers are started, so that there is a parser for each of these parts of the stack head set.

When a parser *P* runs into a token that has already been processed by a subsequent parser *Q*, *P* waits until *Q* gets suspended (or finishes); it then combines the results. If *Q* was split, *P* must wait until all parsers that resulted from this split (and further splits) are either finished or suspended. We will discuss this combination of results in more detail in the example below. Parser *P* may then be able to continue, or it may have to wait again or be suspended. The first parser will never get suspended (though it may have to wait) and will ultimately finish the job.

Now let us examine how this works with the example of Figure 9.18, with input **n-n-(n-n)$**. Let us assume that we cut the input in two pieces of equal length: **n-n-(** and **n-n)$**. Figure 14.1 presents the steps that the first parser takes. It starts

①	**n-n-(**	shift to ③
① **n** ③	**-n-(**	reduce **T → n**
① **T** ②	**-n-(**	reduce **E → T**
① **E** ④	**-n-(**	shift to ⑦
① **E** ④ **-** ⑦	**n-(**	shift to ③
① **E** ④ **-** ⑦ **n** ③	**-(**	reduce **T → n**
① **E** ④ **-** ⑦ **T** ⑧	**-(**	reduce **E → E-T**
① **E** ④	**-(**	shift to ⑦
① **E** ④ **-** ⑦	**(**	shift to ⑥
① **E** ④ **-** ⑦ **(** ⑥		wait

Fig. 14.1. Fischer parsing of the substring **n-n-(**

with one stack head containing the initial state, state ①. It ends up waiting for the second parser, in state ⑥.

To determine the stack heads of the second parser, we have to consider the first token: `n`. There are three states that have a shift on `n`: states ①, ⑥, and ⑦, so our set of stack heads consists of these states. The first few steps of the second parser are presented below.

{ ①, ⑥, ⑦ }	n-n)$	shift to ③
{ ①, ⑥, ⑦ } n ③	-n)$	reduce T → n
{ ①, ⑥ } T ②	-n)$	reduce E → T
⑦ T ⑧	-n)$	reduce E → E-T

The first two steps above actually represent three stack heads but are combined into one line. This is possible as long as the only difference is the state on the bottom of the stacks. Now, state ② requires a reduction for which its stack head is deep enough, but state ⑧ requires a reduction for which it is not, so a split is required. The second part of the split is suspended immediately, so we will now examine what happens with the first part.

{ ①, ⑥ } T ②	-n)$	reduce E → T
① E ④	-n)$	shift to ⑦
⑥ E ⑨	-n)$	shift to ⑦
① E ④ - ⑦	n)$	shift to ③
⑥ E ⑨ - ⑦	n)$	shift to ③
① E ④ - ⑦ n ③	)$	reduce T → n
⑥ E ⑨ - ⑦ n ③	)$	reduce T → n
① E ④ - ⑦ T ⑧	)$	reduce E → E-T
⑥ E ⑨ - ⑦ T ⑧	)$	reduce E → E-T
① E ④	)$	error on). Discarded
⑥ E ⑨	)$	shift to ⑩
⑥ E ⑨) ⑩	$	reduce T → (E)

So, now all splits are either suspended or finished (because of an error), and the suspended stack heads are:

⑥ E ⑨) ⑩	$	reduce T → (E)
⑦ T ⑧	-n)$	reduce E → E-T

These results can now be combined with the result of the first Fischer parser. The state set at the top of the stack of the first Fischer parser (⑥) has no states in common with the bottom state of the second suspended parser (⑦). Therefore, this is a dead end, and is discarded. Combining the result of the first suspended parser with the result of the first Fischer parser, we get:

① E ④ - ⑦ (⑥ E ⑨) ⑩	$	reduce T → (E)
① E ④ - ⑦ T ⑧	$	reduce E → E-T
① E ④	$	shift to ⑤
① E ④ $ ⑤		reduce S → E$
		accept

As we can see from this example, the second part of the input results in a lot of work that later turns out to be without merit, but at least the second part could mostly be processed independently of the first part. However, unfortunate splits are possible. For instance, if an expression is split up right in front of a `)`, the parser processing the part that starts with a `)` can only shift it and then suspend, leaving all the work to its left neighbor, and making the process virtually sequential. So, it is important to split the input in suitable places. There are two ways in which this may be achieved:

- We may split up at terminal symbols that have only a few states with a shift on that symbol. This keeps the number of stack heads small. This may not always be a good heuristic though, see our example of the `)`, for which only state ⑨ has a shift.
- We may also split up in such a way that it is unlikely that the parser will have to suspend itself because it meets a reduce for which its stack is insufficient. For instance, a split right in front of a `(` might be a good heuristic, because the parser processing that part can then at least process its input up until the corresponding closing parenthesis. Parallel bracket matching techniques can be used to find suitable places to split up. See, for instance, Bar-On and Vishkin [225] or Srikant [230].

The time we have to wait for the answer consists of two components: the time required by the N parsers, each working in parallel on n/N of the input ($O(n/N)$) and the time required by the N processes to communicate their findings ($O(N)$); together this is $O(n/N) + O(N)$. This shows that when n is large with respect to N, adding processors helps, but also that after a certain number of processors the $O(N)$ terms will start to dominate and adding further processors will be detrimental. Since the grammar is incorporated in the LR(0) table the speed does not depend on the grammar, but the technique works for LR(0) languages only.

It is interesting to note that the behavior the start-up phase of each parser except the first one is very similar to that of the GLR suffix parser of Bates and Lavie [214] discussed in Section 12.3.2.1. In fact, the Fischer parser parses 1 prefix and $N-1$ substrings, and combines the result.

A (simpler) finite-state variant of Fischer's method can also be used to construct a parallel lexical analyser: each of the N finite-state automata starts in all states, except the first one, which starts in the initial state. Only a few of these states will survive until the end of the chunk that is processed by each automaton. Only these survivors are available for combination into the complete list of resulting tokens.

14.3 Process-Configuration Parsers

A *process-configuration parser* assigns an agent, a process, to each task in the parser. They are also known as *agent parsers*. Agents are autonomous processes whose actions are triggered by messages: when an agent receives a message, this triggers some computation, after which the agent may send messages to other agents. In an

LR parser, for instance, an agent could process a particular state in the LR automaton. State transitions are modeled as messages from one agent to another, so that a message contains enough information for the agent to continue. See, for instance, Hendrickson [235].

14.3.1 A Parallel Bottom-up GLR Parser

An interesting example of a process-configuration parser is presented by Sikkel and Lankhorst [233]. Their "PBT" (Parallel Bottom-up Tomita) parser allocates an agent to each position in the input, one in front of each input symbol, and one at the end. Each agent yields the constituents that start at its own position. The algorithm works with any CF grammar, including ones with ε-rules (hence the agent at the end of the input). The left-to-right restriction of GLR is abandoned, which allows each agent to start parsing at its own word, in parallel. For convenience, an end marker (`#`) is added at the end of the input, so that the last agent has a symbol to work with as well.

We will use the grammar of Figure 7.8 and input `a-a+a#` as a running example, where we have added a rule `S'→S#` to the grammar, and the end marker to the input. As parsing proceeds, an agent P_k will send items that it has found to its left neighbor P_{k-1}. It will also send any items it receives from its right neighbor on to its left neighbor. These items all have the form (i, X, j), where X is either a terminal or a non-terminal, and which indicates that $X \stackrel{*}{\rightarrow} a_i \cdots a_{j-1}$.

For the example, the agents are driven by the parse table of Figure 14.2. Its construction will be discussed later. This example table has a shift/reduce conflict in state 2, but conflicts are allowed. The agent tries all possibilities. All agents are started in state 1.

	action	*goto*							
		`a`	`+`	`-`	`S`	`E`	`F`	`Q`	`#`
1		6	7	8	9	2	5		
2	*reduce* `S→E`							3	
3							4		
4	*reduce* `E→EQF`								
5	*reduce* `E→F`								
6	*reduce* `F→a`								
7	*reduce* `Q→+`								
8	*reduce* `Q→-`								
8	*reduce* `S→E`								
9									*accept*

Fig. 14.2. The PBT parse table for the grammar of Figure 7.8

Let us consider agent P_5, the one dealing with position 5 in the input, in detail. It sees the symbol `a`, so it sends the item `(5,a,6)` to its left neighbor, P_4. Then, it consults the parse table to determine what to do next. The parse table tells it to

shift to state 6, which prescribes a reduce using rule **F→a**. This means that an **F** is recognized, so the agent sends the item **(5,F,6)** to P_4. When applying a reduce, the parser keeps the old stack around, because it may still be needed. It uses stack duplication, with combined prefixes, as described in Section 11.1.2.3. After the reduce, the parser is back in state 1, and sees an **F**. The parse table instructs the parser to shift to state 5, which again prescribes a reduce, now using rule **E→F**. So, an **E** is recognized, and the item **(5,E,6)** is sent to P_4. Another shift, another reduce, a new item **(5,S,6)** sent to P_4, and another shift bring P_5 into state 9. Somewhere along the way, agent P_6 has discovered the **#** and sent the item **(6,#,7)** to P_5. Agent P_5 now forwards this item to P_4, and also determines that the sentence starting at position 5 is accepted. The final stack of P_5 is depicted in Figure 14.3.

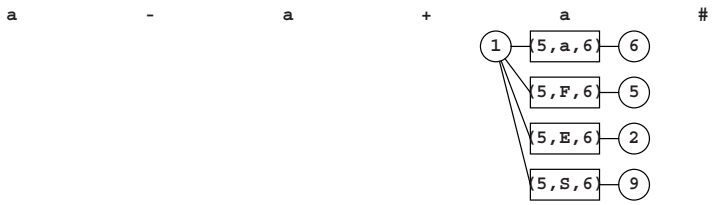


Fig. 14.3. The stack of agent P_5

The reader is invited to verify that P_4 derives the item **(4,Q,5)**. It also sends the items found by P_5 through to P_3. Now, let us look at P_3 in more detail. After its local processing of its own symbol, its stack looks very much like the one from Figure 14.3, with some positional differences. Next, it receives the **(4,Q,5)** item from P_4. P_3 has a stack at position 4, in state 2, in which a **Q** can be shifted. The resulting stack is shown in Figure 14.4.

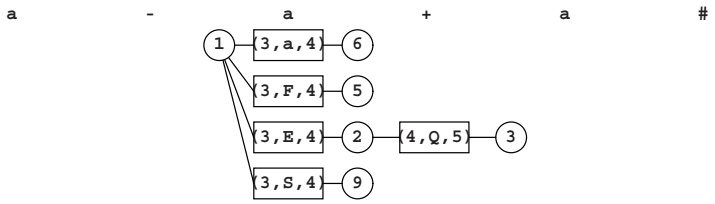


Fig. 14.4. The stack of agent P_3 after shifting Q

Next, it receives the item **(5,F,6)** and this **F** can now be shifted. There is a subtle issue here concerning the order of the messages: before passing on items that start at a position k, an agent must send all items that end right in front of this position and that pass through this agent. Shifting the **F** has brought P_3 into state 4, which prescribes a reduce using rule **E→EQF**. So, P_3 has discovered the item **(3,E,6)**.

Continuing, it also discovers `(3,S,6)` and also discovers that the sentence starting at position 3 is accepted. The final stack of P_3 is shown in Figure 14.5.

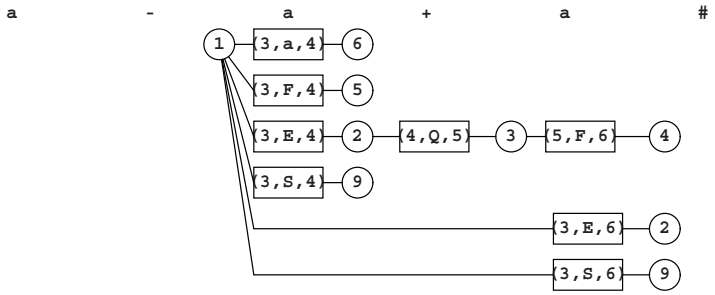


Fig. 14.5. The final stack of agent P_3

Like agent P_4, agent P_2 only derives the item `(2,Q,3)`, so we finally turn our attention to P_1. Initially, P_1 develops its stack quite similarly to P_3. However, where P_3 finishes, P_1 can continue. It has a stack head in state 2 after processing 3 symbols. When it receives the item `(4,Q,5)` from P_2, shifting the `Q` brings it into state 3, and when it then receives `(5,F,6)`, shifting the `F` brings it into state 4, which, as we know by now, prescribes a reduce using rule `E→EQF`. So, P_1 has discovered the item `(1,E,6)`. It also discovers `(1,S,6)`, ends up in state 9, where `(6,#,7)` causes P_1 to accept the sentence. See Figure 14.6.

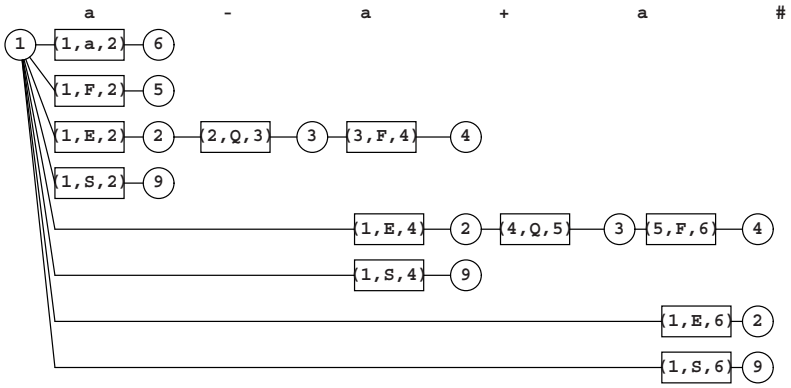


Fig. 14.6. The final stack of agent P_1

Some observations should be made here:

- All states on the stack are active, not just the ones on top of a stack head. Each state can start a branch when an item arrives that can be shifted in this state.

- The order in which the items are sent is important. When an agent P_i tries to fit an item (j,X,k) on any one of the states on its stack, all items (l,Y,m) with $i \leq l \leq m \leq j$ must have been processed, which is a bit of a problem when $l = m = j = k$. Nullable symbols have to be retried.
- Although the stack of each agent is in fact tree structured, with common prefixes, the complete conceptual picture of the stack is graph structured: every state transition labeled (i,X,j) implicitly refers to agent P_i which has the details for (i,X,j).
- The number of items sent from one agent to another can be reduced considerably. For instance, if a symbol (either terminal or non-terminal) X only occurs as a first symbol in the right-hand sides of the productions, and agent P_i has discovered the item (i,X,j), this is only useful information for agent P_i. Also, if P_i discovers $(i+1,Y,j)$, then this information is only useful for other agents if the combination a_iY occurs in a right-hand side but not at the beginning, or a combination AY occurs in a right-hand side and produces a string ending with a_i.

Generation of the parse table is particularly easy. It is similar to the LR(0) parse table generation, but there are some differences: since each agent is initially ready to recognize any non-terminal, the initial state contains all items $A \rightarrow \bullet\alpha$ for all grammar rules $A \rightarrow \alpha$ in the grammar. Next, for each symbol X after a dot, a new state is defined with the items that have the dot after the X. The resulting automaton is deterministic by the way it is constructed (but it may contain shift/reduce conflicts). For our example, the automaton is shown in Figure 14.7.

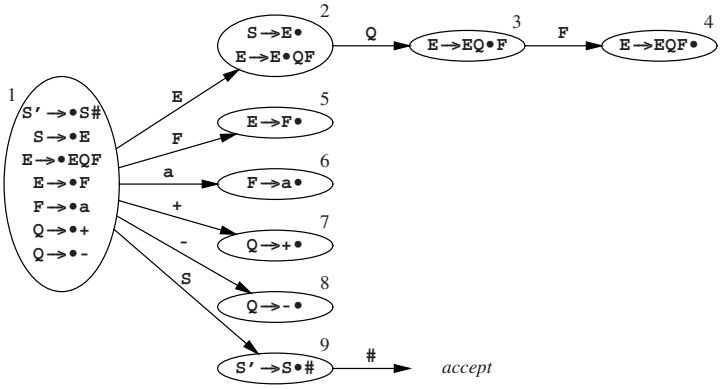


Fig. 14.7. The PBT automaton for the grammar of Figure 7.8

To obtain a parse-forest grammar, each agent must make a grammar rule for each reduction it finds. For instance, when agent P_i finds that it can reduce (i,Y,j) and (j,Z,k) to (i,X,k), it adds a rule $X\_i\_k \rightarrow Y\_i\_jZ\_j\_k$. It can be sure that if Y is a non-terminal, it has rules for Y_i_j, because it has made a reduction to it at some time. Likewise, if Z is a non-terminal, agent P_j has made a reduction to Z_j_k

and thus has one or more grammar rules for it. At the end of the parsing process, we have a parse-forest grammar, but it is distributed over the agents. One way of dealing with this is to send the parse-forest grammar rules with the items. This has the disadvantage that items may now have to be sent several times (with different grammar rules).

Sikkel and Lankhorst [233] do not give a theoretical derivation of time and communication requirements of their algorithm. Rather, they did a practical performance evaluation that showed that for large sentences, their algorithm actually gave a speedup of about $O(\sqrt{n})$ when using n processors with respect to a sequential Tomita parser.

14.3.2 Some Other Process-Configuration Parsers

Yonezawa and Ohsawa [229] describe an agent parser where there is one agent for each grammar rule. The agent for the rule $N \rightarrow ABC$ receives messages from all agents that manage rules for A, B and C and sends messages to all agents for rules of the form $P \rightarrow \alpha N \beta$. Each message is a parse tree for a chunk of the input, including its position in the input and its length. The agent waits for chunks of the right nonterminal and of the right position and length, combines them into a new parse tree and sends it to the interested agents. In the end, the agent processing the start symbol delivers all parse trees. The parser does not allow ε-rules or circularities.

Ra and Kim [236] describe what they call an Earley-based parser, which looks like an Earley parser without the top-down component, i.e., a bottom-up Earley parser, but which is actually a bottom-up left-corner parser (see Section 7.3.4 and Sikkel [158]). Their parser too does not allow ε-rules or circularities. It bears resemblance to the PBT parser of Section 14.3.1 in that it has an agent for each position in the input, and sends the completed items that it finds to its left neighbor. However, it does not use a parse table. Instead, each agent maintains a set of dotted items, which have the format $(i, A \rightarrow \alpha \bullet \beta, j)$; this means that we have recognized an α in positions i up to (but not including) j and are looking forward to recognizing a β starting at position j. Each agent P_i initializes its item set with all items $(i, A \rightarrow \bullet \alpha, i)$ for all grammar rules $A \rightarrow \alpha$. Next, each agent P_i performs the following steps, repeatedly.

- A "scanner" step: if the agent has an item $(i, A \rightarrow \alpha \bullet a \beta, j)$ and $a = a_j$, it adds the item $(i, A \rightarrow \alpha a \bullet \beta, j+1)$.
- A "starter" step: for each item $(i, A \rightarrow \alpha \bullet, j)$ (with the dot at the end of the rule, which means that the rule has been recognized completely) and each grammar rule $B \rightarrow A\beta$, the item $(i, B \rightarrow A \bullet \beta, j)$ is added. This corresponds to the left-corner inference rule of Section 7.3.4. As β can be ε, in which case a new completely recognized item has been found, this step has to be performed repeatedly until no new items are added.
- An "extender" step: if the agent has an item $(i, A \rightarrow \alpha \bullet B \beta, r)$ for some r and a received message contains the item (r, B, j), the agent adds the item $(i, A \rightarrow \alpha B \bullet \beta, j)$ to its item set.

Initially, the agents process their own itemset, until no new items can be added. Then, each received item may trigger new additions. Each new addition that represents a completed item is passed on to the left neighbor, as are all items that are received from the right neighbor. Conceptually, the process can be divided into stages: after stage k (in which agent P_i computes "its" items of length k), P_{i+1} sends its own completed items of length k, the completed items from P_{i+2} of length $k-1$, the completed items from P_{i+3} of length $k-2$, et cetera. So, messages may become bigger with k, but the number of messages decreases: after the initialization, agent P_n will never receive messages, agent P_{n-1} will only receive one message, et cetera. In the end, when agent P_1 has processed $n-1$ messages, it has computed its items of length n. If this item set now contains the item $(1, S \rightarrow \alpha\bullet, n+1)$, where S is the start symbol of the grammar, the sentence is recognized.

Ra and Kim [236] present a time requirement analysis, and conclude that the worst-case performance is $O(n^3/p)$ on p processors. It should be noted, however, that there is also a dependency on the size of the grammar that is at least quadratic.

We can see that the above algorithms are all parallel ways of filling the items table of Figure 7.13.

14.4 Connectionist Parsers

A connectionist parser is a parser that runs on a connectionist network. In general, a *connectionist network* is a network of nodes connected by unidirectional lines which each carry a value, the level of activation, which is determined by the node the line emanates from. Each node continually examines the activity levels on its input lines and computes from them the activity level on its output line(s).

To allow for reasoning about time, we split time up into discrete steps, and assume that it takes one time step to compute the activity level on the output lines from the activity levels on the input lines. In other words, the activity level on the output lines at time t is computed from the activity level on the input lines at time $t-1$.

14.4.1 Boolean Circuits

The simplest connectionist network is a *Boolean circuit*, which is a directed graph where the nodes of the graph correspond to the nodes in the connectionist network, and the (directed) edges of the graph correspond to the connections in the network. In a Boolean circuit, there are only two activation levels: "on" and "off", and there are only two types of nodes: "OR-nodes" and "AND-nodes". Each node has an arbitrary number of inputs and an arbitrary number of outputs (the outputs all carry the same value). An OR-node determines its output value as follows: if any of its input values is "on", its output value will be "on", otherwise it will be "off". An AND-node determines its output value as follows: if any of its input values is "off", its output value will be "off", otherwise it will be "on". We say that a node is "off" when its output value is "off", and it is "on" when its output value is "on". Usually, Boolean circuits may have NOT-nodes as well. A NOT-node is a node with a single input

and a single output, where if the input is "on", the output is "off", and vice versa. However, we do not need NOT-nodes for the discussion below.

The circuit is started by setting the initial activation level for some of the nodes to "on". At each time step, the outputs of a node are computed as the result of the node computation with the input values of the previous time step. This means that if there are no changes in a single time step, the input values of the next time step are the same, and thus the output values are too, so the system is stabilized. This happens after a number of time steps, because once a node is "on", it will never be "off" again. When the circuit has no cycles, the number of time steps required to propagate the initial values through the whole graph is equal to the number of connections in the longest path from any initial node.

Fig. 14.8. A circuit representing $d = (a \text{ AND } b) \text{ OR } c$

Figure 14.8 presents a small example of a Boolean circuit, with an AND-node computing the result of a AND b, and an OR-node with inputs from the AND-node and c. The result d is "on" if either c is "on" or both a and b are "on". The circuit takes two timesteps to stabilize. When a Boolean circuit has NOT-nodes, it is possible for such a circuit to never stabilize.

14.4.2 A CYK Recognizer on a Boolean Circuit

A CYK recognizer is particularly easy to implement with a Boolean circuit, as long as we limit the length of the sentence to some upperbound. Suppose we want to build a CYK recognizer for a grammar G and input sentences with length at most N. As we did in Section 4.2.2, we assume that the grammar is in Chomsky Normal Form. [1]

The idea is that we assign a node to every possible hypothesis "non-terminal A derives substring $s_{i,l}$, starting at position i and with length l". We will call this node $A_{i,l}$. If this hypothesis is realized, A is a member of the set $R_{i,l}$, as discussed in Section 4.2.2. So, each set $R_{i,l}$ has a node for every non-terminal, and its output is "on" if the non-terminal is a member of the set, and "off" otherwise. Note that in total there are $O(n^2|V_N|)$ of these hypotheses (and nodes), where $|V_N|$ stands for the number of non-terminals.

Next, we will determine what kind of nodes these are, and what their inputs are. For a non-terminal A to derive a substring $s_{i,l}$, there must be at least one rule $A \rightarrow BC$ that derives this substring.

For a rule $A \rightarrow BC$ to derive a substring $s_{i,l}$, the right-hand side BC must derive this substring. This means that for some k, B must derive $s_{i,k}$ and C must derive

[1] The Boolean circuit implementation presented here is by Sikkel [158].

$s_{i+k,l-k}$. So, if we have a node $B_{i,k}$ that represents the hypothesis that B derives $s_{i,k}$ and we have a node $C_{i+k,l-k}$ that represents the hypothesis that C derives $s_{i+k,l-k}$, the combined hypothesis can be represented by an AND-node $A \rightarrow BC_{i,k,l}$ with two inputs: the outputs of nodes $B_{i,k}$ and $C_{i+k,l-k}$. For every length k between 1 and $l-1$ we create such a node. Now, we see that node $A_{i,l}$ will be an OR-node with inputs from all nodes $A \rightarrow BC_{i,k,l}$.

This tells us how to deal with rules of type $A \rightarrow BC$, and describes the main engine, but we still have to initialize and start it. For that purpose, we have AND-nodes a_i for all terminals a, for all positions i. These nodes have two inputs, one that is used to start the circuit (when it is turned "on" by a switch or so), and one that is "on" if, in fact, $s_{i,1}$ is a. We also have AND-nodes $A \rightarrow a_i$ for all production rules $A \rightarrow a$, for all positions i, which get their one input from the AND-node a_i. This node is turned on when the symbol in position i is, in fact, a. Finally, we also have OR-nodes $A_{i,1}$ for all non-terminals A and all positions i. These nodes represent the hypotheses that the non-terminal A derives the substring $s_{i,1}$. Their inputs are the outputs of all nodes $A \rightarrow a_i$.

In the end, when the circuit is stabilized, we look at the node representing the hypothesis $S_{1,n}$, that is, the hypothesis that the start symbol S derives the input sentence $s_{1,n}$. When this node is "on", the sentence is recognized, when it is "off", it is not. With the circuit constructed above we also can recognize sentences of length less than the maximum length N. Like we saw with a CYK parser, where a sentence $s_{1,k}$ is recognized when S is a member of $R_{1,k}$, in the Boolean circuit, the node $S_{1,k}$ will be turned "on".

Time for an example! We will use grammar 14.9 as an example to build a Boolean circuit from. We will build a recognizer for this language for strings with length at most 3, because things get out of hand rather quickly.

```
S_S  →  D B
S_S  →  A S
S_S  →  c
 D   →  A S
 A   →  a
 B   →  b
```

Fig. 14.9. An example grammar to build a Boolean circuit

Figure 14.10 presents the Boolean circuit recognizer for the grammar of Figure 14.9. The circuit is started by flipping the switch at the bottom right hand side of the picture. The arrows leaving the picture at the left are the lines to check when the circuit is stabilized. When $S_{1,1}$ is "on", a sentence of length 1 is recognized, et cetera. The elements of the CYK recognition table $R_{i,j}$ are represented by the dotted boxes.

We will follow the time steps on recognizing the input sentence **acb**, which is represented by setting the left inputs of nodes a_1, c_2, and b_3 to "on". Now, when we flip the switch, the right inputs of nodes a_1, c_2, and b_3 are also "on". Therefore, one timestep after flipping the switch, the inputs of $A \rightarrow a_1$, $S \rightarrow c_2$, and $B \rightarrow b_3$

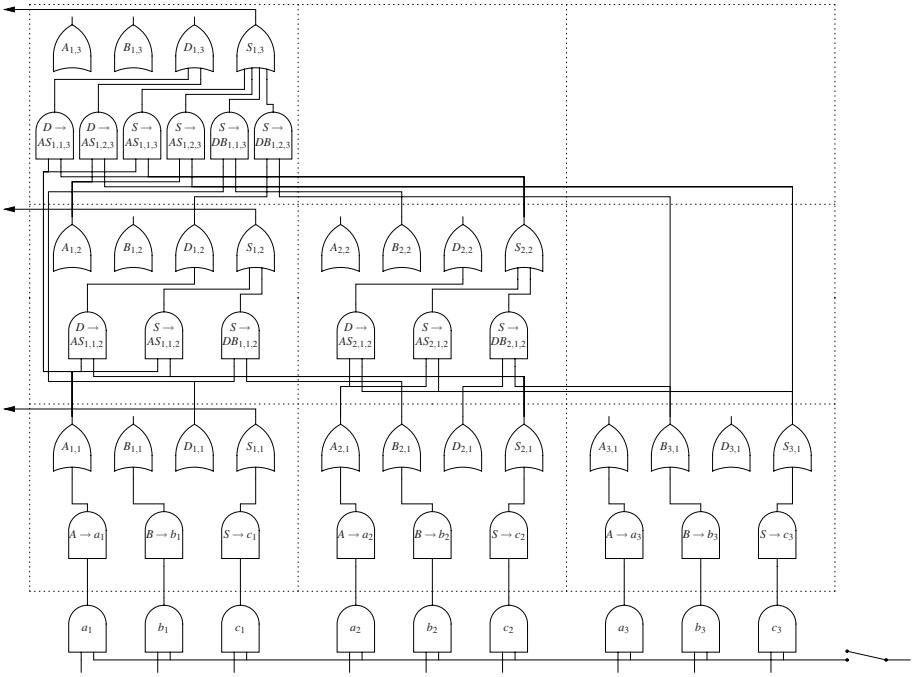

$A_{1,3}$ $B_{1,3}$ $D_{1,3}$ $S_{1,3}$
$D \to AS_{1,1,3}$ $D \to AS_{1,2,3}$ $S \to AS_{1,1,3}$ $S \to AS_{1,2,3}$ $S \to DB_{1,1,3}$ $S \to DB_{1,2,3}$
$A_{1,2}$ $B_{1,2}$ $D_{1,2}$ $S_{1,2}$ $A_{2,2}$ $B_{2,2}$ $D_{2,2}$ $S_{2,2}$
$D \to AS_{1,1,2}$ $S \to AS_{1,1,2}$ $S \to DB_{1,1,2}$ $D \to AS_{2,1,2}$ $S \to AS_{2,1,2}$ $S \to DB_{2,1,2}$
$A_{1,1}$ $B_{1,1}$ $D_{1,1}$ $S_{1,1}$ $A_{2,1}$ $B_{2,1}$ $D_{2,1}$ $S_{2,1}$ $A_{3,1}$ $B_{3,1}$ $D_{3,1}$ $S_{3,1}$
$A \to a_1$ $B \to b_1$ $S \to c_1$ $A \to a_2$ $B \to b_2$ $S \to c_2$ $A \to a_3$ $B \to b_3$ $S \to c_3$
a_1 b_1 c_1 a_2 b_2 c_2 a_3 b_3 c_3

Fig. 14.10. A recognizer for the grammar of Figure 14.9

are "on", and one timestep later the inputs of $A_{1,1}$, $S_{2,1}$, and $B_{3,1}$ are "on". After the third timestep, the inputs of $S \to AS_{1,1,2}$ and $D \to AS_{1,1,2}$ are "on". This, in turn, will set the inputs of $D_{1,2}$ and $S_{1,2}$ "on" at the next timestep. So, the sentence **ac** is recognized, but we are not done yet. Both inputs of $S \to DB_{1,2,3}$ are now "on", so its output will be "on" at the next timestep, and $S_{1,3}$ will be "on" next, which means that the sentence is recognized.

As can be seen from Figure 14.10, the circuit as constructed above contains many nodes that can never be activated. Its size can be reduced using *metaparsing*: we let the circuit try and recognize any input sentence of length $\leq N$ by turning on all nodes a_i. Any node not turned "on"eventually by this input can be discarded, because if it is not turned "on" now, it will not be turned on, ever. This results in the recognizer as presented in Figure 14.11.

Next we mark all nodes reachable by a *reversed scan*, starting with nodes $S_{1,k}$ and following all connections in the reversed direction. Any intermediate node not marked in this way can be removed, because there is no path from such a node to a node $S_{1,k}$, see Figure 14.12.

14.4.2.1 Time and Node Requirements

Now let us examine the time requirements for such a Boolean circuit, i.e., how many time steps it takes for the circuit to stabilize. From the description above, it appears

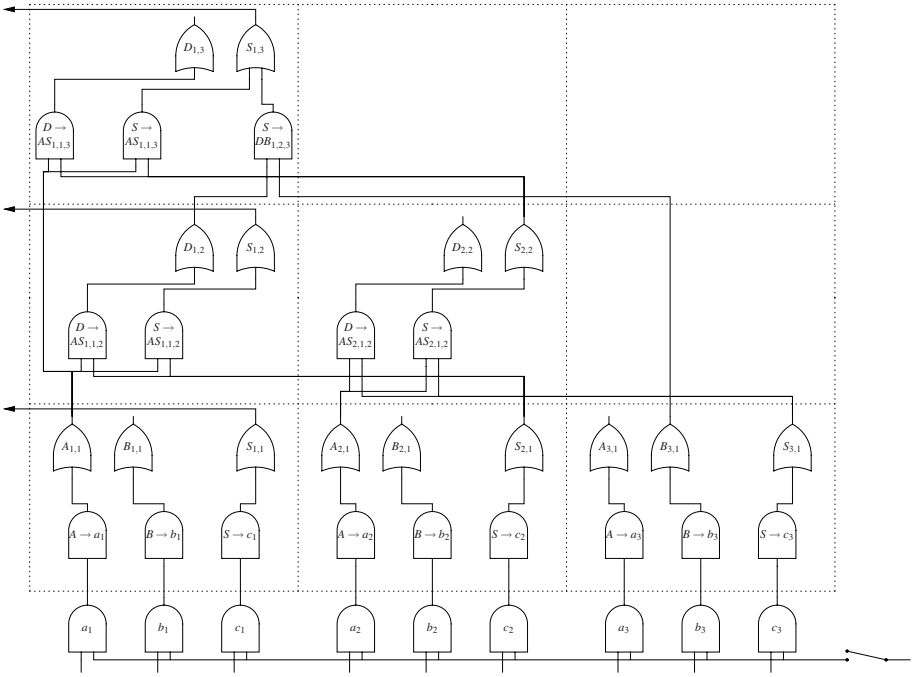


Fig. 14.11. The recognizer of Figure 14.10 after metaparsing

that there are no cycles in the resulting circuit, and getting from nodes associated with length l to nodes associated with length $l+1$ takes two time steps; so getting from nodes associated with length 1 to nodes associated with length n takes $2n$ time steps. In other words, we need $O(n)$ time steps for the system to stabilize; note that this time is independent of the grammar size. This is a major improvement over the $O(n^3)$ that we saw in Section 4.2.2. However, this improvement comes with considerable costs: the number of nodes needed.

Obviously, the number of nodes needed depends on the maximum length of the input sentence. There are $n(n+1)/2$ sets $R_{i,l}$, and for each of these sets, we need:

- an OR-node for each non-terminal. The number of such nodes within one $R_{i,l}$ set thus depends linearly on the number of non-terminals $|V_N|$ in the grammar;
- AND-nodes $C \rightarrow AB_{i,k,l}$ for all rules $C \rightarrow AB$ in the grammar and for all $1 \leq k < l$ (when $l > 1$), or AND-nodes $A \rightarrow a_i$ for all grammar rules $A \rightarrow a$ when $l = 1$. The number of such nodes depends not only linearly on the number of rules $|P|$ in the grammar, but also linearly on l, so ultimately also on the maximum length of the input sentence.

Thus, considering the number of non-terminals, the number of grammar rules, and the number of terminals constant, the number of nodes needed for a single set $R_{i,l}$ is $O(n|G|)$ where $|G|$ is the size of the grammar, so the total number of nodes needed for R is $O(n^3|G|)$. In addition, we need $n|\Sigma|$ AND-nodes, where $|\Sigma|$ stands for

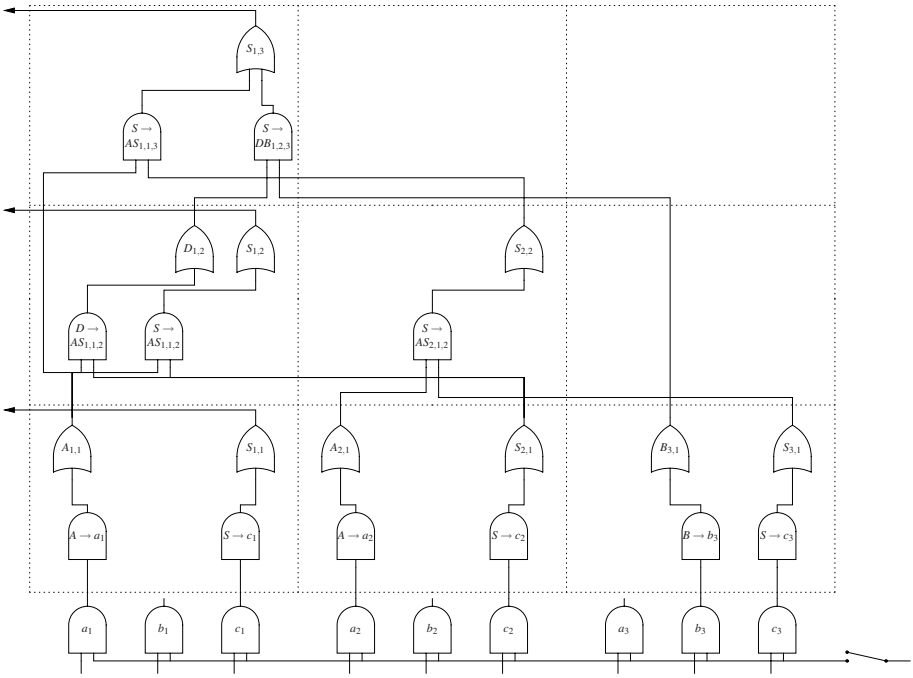


Fig. 14.12. The recognizer of Figure 14.11 after a reversed scan

the number of different terminals (Σ is the set of terminal symbols). This number is $O(n|G|)$, so the total number of nodes needed is $O(n^3|G|)$ for parsing in time $O(n)$. We see that although the time requirement is independent of $|G|$, $|G|$ reappears in the node requirements. Unlike Fischer's parser, adding more processors to a connectionist parser to speed it up is not an option: G and n together determine exactly the number of processors the parser needs.

Chang, Ibarra and Palis [228] show that a linear-time CYK parser can be implemented on $O(n^2)$ processors, a two-dimensional array of finite-state machines.

It is interesting that the above connectionist parser can be seen as a process-configuration parser in which the connections are hard-wired.

14.4.2.2 Creating a Parse Forest

One might think that the circuit as presented above, once it is stabilized, represents a parse forest, but in general this is not the case: outputs and nodes can be turned "on" but not participate in a valid parsing. This corresponds to the introduction of non-reachable rules when getting a parse-forest grammar from a CYK recognition table, as we have seen in Section 4.2.8. There, we had to remove non-reachable rules from the parse-forest grammar. In Boolean circuit terms, this means that we have to make sure that a node that is turned "on" contributes to the ultimate turning "on" of $S_{1,N}$.

On the other hand, all nodes that are left in the circuit after the reversed scan could represent nodes in a parse forest. The problem is that we need a top-down scan to determine which nodes should actually be turned "on". This requires the value of node $S_{1,N}$ as input, which means that we have to build additional circuitry for the parse forest. We basically need additional circuitry that determines which nodes actually are part of the parse forest, and which nodes are non-reachable for the input at hand. The additional circuitry will have a node X' for every node X in the recognizing circuit. X' will be an OR-node if X is, and an AND-node if X is. X' will be turned "on" if, and only if, it is part of the parse forest. The difference with the recognizing circuitry lies in the wiring (the connections), so we will now concentrate on that.

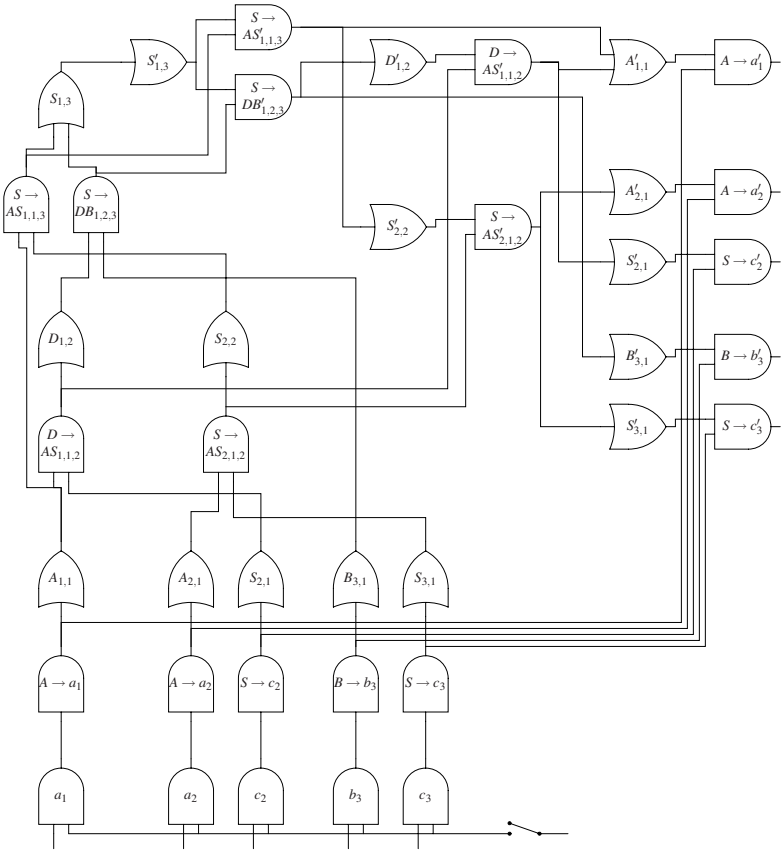


Fig. 14.13. The recognizer/parse-forest circuit for the grammar of Figure 14.9

Starting with the top level, as we would do when marking the reachable rules in the parse-forest grammar, where we would mark S_1_N reachable, the node $S'_{1,N}$, an

OR-node, gets a single input: the output of $S_{1,N}$. This can be considered a switch to turn on the parse-forest circuitry. Next, we turn our attention to all nodes $C \rightarrow AB_{i,j,k}$ which determine the inputs of node $C_{i,k}$. Node $C \rightarrow AB'_{i,j,k}$ should be part of the parse forest if $C \rightarrow AB_{i,j,k}$ is "on" (obviously), and $C'_{i,k}$ is "on" (which makes sure that it is actually part of the parse forest). So, $C \rightarrow AB'_{i,j,k}$ is an AND-node with two inputs: $C \rightarrow AB_{i,j,k}$ and $C'_{i,k}$. It has two outputs: one to node $A'_{i,j}$ and one to $B'_{i+j,k-j}$. This corresponds to marking *A_i_j* and $B\_i + j\_k - j$ reachable when *C_i_k* is and the rule is applicable for the input at hand (see Section 4.2.8).

In addition, for nodes $A'_{i,1}$ we add an output to node $A \rightarrow a'_i$, which is an AND-node with two inputs, for all rules $A \rightarrow a$. The second input to node $A \rightarrow a'_i$ comes from the output of node $A \rightarrow a_i$.

The complete circuit for our example grammar and strings of length 3 is presented in Figure 14.13. To keep the figure readable, we have removed the nodes that do not contribute to the recognizing of strings of length 3 (again, using a reversed scan).

14.4.3 Rytter's Algorithm

Many problems that require linear or more time on a sequential processor can be solved in $O(\log\ n)$ time on a parallel machine, so the question arises if parallel parsing in $O(\log\ n)$ is possible. Rytter [226, 227] invented a clever recognition algorithm that can be executed in $O(\log\ n)$ time. We will first describe the building blocks of the algorithm, then describe the algorithm itself, and then show how to build a Boolean circuit for it. We will use the grammar of Figure 14.9 as a running example, with input sentence **aacb**. Virtually the same algorithm was invented by Brent and Goldschlager [224]. In fact, their paper was published earlier.

In this discussion, we will use the version of Brent and Goldschlager, but call the algorithm Rytter's algorithm, for that is the name it goes by.

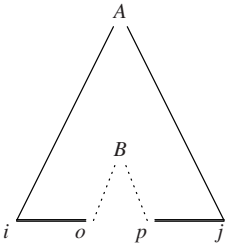
Like the CYK algorithm, Rytter's algorithm requires the grammar to be in Chomsky Normal Form. Also like CYK, Rytter's algorithm maintains a set of hypotheses that have been realized. Like CYK, this set is initialized with all hypotheses of the form "non-terminal A derives substring $s_{i,1}$", where $A \rightarrow s_{i,1}$ is a rule of the grammar, and $s = s_1 \cdots s_n$ is the input sentence, and $s_{i,1}$ is the substring starting at position i, of length 1. We will denote CYK hypotheses with a triple (A, i, j), meaning that the hypothesis is: "non-terminal A derives substring $s_{i+1} \cdots s_j$". Note that we have switched from length to index, which is more convenient for the following description. We will call the set of realized CYK hypotheses S_{CYK}.

For our example, initially, S_{CYK} consists of the following elements:

```
(A,0,1)  (A,1,2)  (S,2,3)  (B,3,4)
```

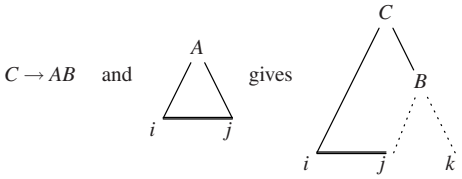
Now, in the CYK algorithm, if there is a grammar rule $C \rightarrow AB$, and we have two realized CYK hypotheses "(A, i, j)" and "(B, j, k)", the CYK hypothesis "(C, i, k)" will be realized. In this way, the CYK algorithm maintains a set of realized CYK hypotheses. The Rytter algorithm, in addition, maintains a set of what we will call *Rytter*

proposals. The general form of a Rytter proposal consists of two CYK hypotheses, denoted as follows: $(A,i,j;B,o,p)$, and it is realizable if $A \stackrel{*}{\rightarrow} s_{i+1} \cdots s_o B s_{p+1} \cdots s_j$. We will call a Rytter proposal realized if we have determined that it is, in fact, realizable.

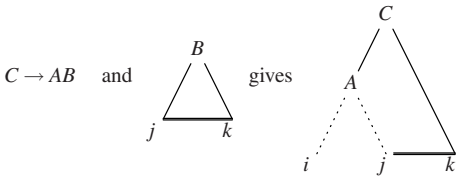


In fact, a Rytter proposal looks like a CYK hypothesis with a gap. The gap is speculative: if this Rytter proposal is realized, and the CYK hypothesis (B,o,p) is realized, then (A,i,j) will be realized. This is just logic: if U implies V, and U is true, then V is true.

The first building block of the algorithm is what we call the "propose phase". Based on the grammar rules and the current set of CYK hypotheses, a Rytter proposal is constructed as follows: if we have a grammar rule $C \rightarrow AB$ and S_{CYK} contains a hypothesis (A,i,j), then we know that if the CYK hypothesis (B,j,k) is realized, the hypothesis (C,i,k) will also be realized. We can formulate this as a speculation: (C,i,k) could be realized, in the case that (B,j,k) turns out to be realized. This is the Rytter proposal $(C,i,k;B,j,k)$, which now turns out to be realized. It is depicted below, where the speculative part is indicated by dashed lines. Such a speculation can be made for any k such that $j < k \leq n$.



Likewise, with the same grammar rule, if S_{CYK} contains a hypothesis (B,j,k), then we know that if the CYK hypothesis (A,i,j) turns out to be realized, the hypothesis (C,i,k) will also be realized. Again we can formulate this as a speculation: (C,i,k) could be realized, in the case that (A,i,j) turns out to be realized, and we can denote this as follows: $(C,i,k;A,i,j)$. This speculation can be made for any i such that $0 \leq i < j$.

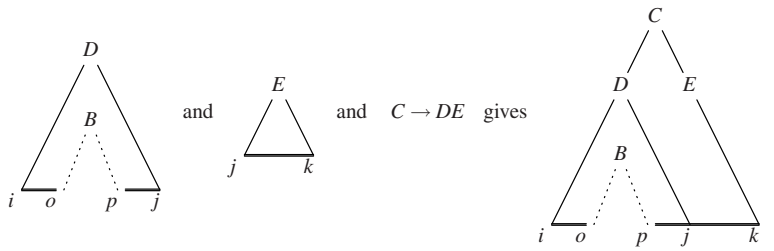


This completes the propose phase. In general, there are many possible Rytter proposals. For instance, the hypotheses in the initial S_{CYK} set give rise to the following realized Rytter proposals:

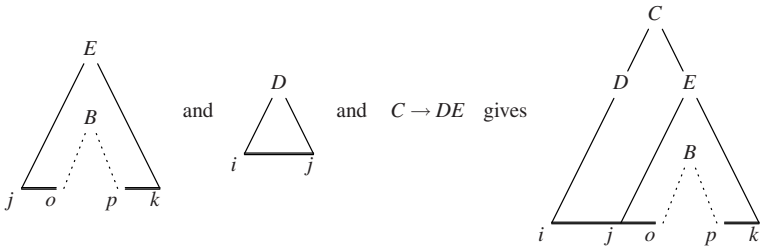
```
(D,0,2;S,1,2)  (D,0,3;A,0,2)  (D,0,3;S,1,3)
(D,0,4;S,1,4)  (D,1,3;A,1,2)  (D,1,3;S,2,3)
(D,1,4;S,2,4)  (S,0,2;S,1,2)  (S,0,3;A,0,2)
(S,0,3;S,1,3)  (S,0,4;D,0,3)  (S,0,4;S,1,4)
(S,1,3;A,1,2)  (S,1,3;S,2,3)  (S,1,4;D,1,3)
(S,1,4;S,2,4)  (S,2,4;D,2,3)
```

We will call the set of realized Rytter proposals S_{Rytter}.

The clever thing about Rytter proposals is that they can be combined, with CYK hypotheses, but also with other Rytter proposals. Brent and Goldschlager [224] combine Rytter proposals with CYK hypotheses to construct new Rytter proposals: if there is a grammar rule $C \rightarrow DE$ and both the Rytter proposal $(D,i,j;B,o,p)$ and the CYK hypothesis (E,j,k) are realized, then $(C,i,k;B,o,p)$ will be realized.



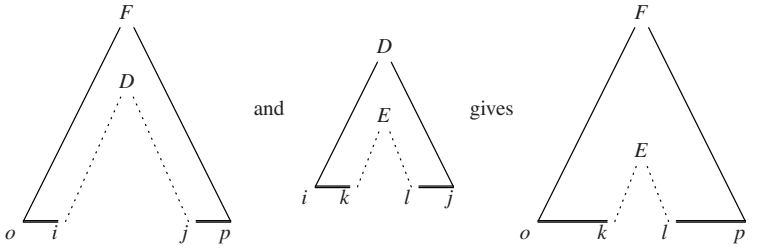
Likewise, with the same grammar rule, if both $(E,j,k;B,o,p)$ and (D,i,j) are realized, $(C,i,k;B,o,p)$ will be realized.



In our example, this gives the following realized Rytter proposals:

```
(D,0,3;A,1,2)  (D,0,3;S,2,3)  (D,0,4;D,1,3)
(D,0,4;S,2,4)  (D,1,4;D,2,3)  (S,0,3;A,1,2)
(S,0,3;S,2,3)  (S,0,4;A,0,2)  (S,0,4;D,1,3)
(S,0,4;S,1,3)  (S,0,4;S,2,4)  (S,1,4;A,1,2)
(S,1,4;D,2,3)  (S,1,4;S,2,3)
```

Rytter proposals can be combined too, as follows: If we have two realized Rytter proposals $(D,i,j;E,k,l)$ and $(F,o,p;D,i,j)$, we can construct a new Rytter proposal $(F,o,p;E,k,l)$, which is then realized. Again, this is just logic: if A implies B, and B implies C, then A implies C, so if (E,k,l) implies (D,i,j), and (D,i,j) implies (F,o,p), then (E,k,l) implies (F,o,p).



In Rytter's description, only Rytter proposals are combined.

This combination of Rytter proposals (also with CYK hypotheses) is the second building block, and we call it the "combine phase". This phase computes a new set of realized Rytter proposals, by first trying all possible combinations of Rytter proposals in S_{Rytter} with CYK hypotheses in S_{CYK}, adding the result to S_{Rytter}, and then trying all possible combinations of Rytter proposals in S_{Rytter}. In the end, this set too is added to S_{Rytter}.

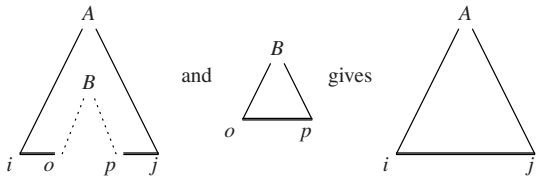
In our example `(D,0,4;D,1,3)` (which was found above) and `(D,1,3;A,1,2)` (which was found in the propose phase) can, for instance, be combined to `(D,0,4;A,1,2)`. To the set of realized Rytter proposals above the following combinations can be added:

```
(D, 0, 4; A, 1, 2)  (D, 0, 4; D, 2, 3)  (D, 0, 4; S, 2, 3)
(S, 0, 4; A, 1, 2)  (S, 0, 4; D, 2, 3)  (S, 0, 4; S, 2, 3)
```

Brent and Goldschlager [224] include a third stage in the combine phase: a CYK combine step on S_{CYK} set, as we saw earlier Section 14.4.2. Rytter does not include such a phase. In our example, this stage results in the addition of the following CYK hypotheses to S_{CYK}:

```
(D,1,3)  (S,1,3)
```

Perhaps not surprisingly, the third building block consists of the combination of Rytter proposals with CYK hypotheses. If we have a realized Rytter proposal $(A,i,j;B,o,p)$ and the CYK hypothesis (B,o,p) turns out to be realized, then the CYK hypothesis (A,i,j) will also be realized.



All combinations of Rytter proposals in S_{Rytter} and CYK hypotheses in S_{CYK} are tried. In the end, all CYK hypotheses that turn out to be realized are added to S_{CYK}. This is called the "recognition phase".

In our example, the presence of `(D,0,3;A,1,2)` in S_{Rytter} and `(A,1,2)` in S_{CYK} results in the CYK hypothesis `(D,0,3)` turning out to be realized. From the initial S_{CYK}, and all proposals in S_{Rytter}, we can now determine that the following CYK hypotheses turn out to be realized:

```
(D,0,3)  (D,0,4)  (S,0,3)  (S,0,4)  (S,1,4)
```

Now that we have the building blocks, we are ready to describe the algorithm itself. After initialization of the S_{CYK} set, as described above, the following sequence of phases is performed repeatedly:

- the proposal phase, which takes the current S_{CYK} and adds the Rytter proposals that can be derived from it to S_{Rytter};
- the combine phase, which tries all possible combinations of proposals in S_{Rytter} and hypotheses in S_{CYK}, adds the newly found proposals to S_{Rytter}, and then tries all possible combinations of proposals in S_{Rytter}, and again adds the newly found proposals to S_{Rytter}; also, realized CYK hypotheses are combined to find new realized CYK hypotheses, which are in the end placed in S_{CYK}.
- and the recognition phase, which combines hypotheses in S_{CYK} with proposals in S_{Rytter} to find new realized CYK hypotheses, which are in the end placed in S_{CYK}.

Now the question arises when to stop this repetition. The answer to that is easy: when a sequence of phases does not deliver any new realized Rytter proposals and no new CYK hypotheses, the next sequence will not deliver any new items either. This will happen at some point, because the number of CYK hypotheses and Rytter proposals is finite. So, the system stabilizes at some point, which can easily be detected.

In the end, when S_{CYK} contains $(S,0,n)$ where S is the start symbol of the grammar and n is the length of the input sentence, the sentence is recognized. In fact, this may also be an added stop criterion: we can also stop the repetition when S_{CYK} contains $(S,0,n)$. In this case, the sentence is recognized. This may suffice when a recognizer is all that is needed. For a parser, however, this is not a good stop-criterion, because some parsings may be missed.

Pursuing the example above, we see that the sentence is already recognized after one iteration, but the system is not quite stabilized yet: the propose phase of the second iteration adds the following Rytter proposals:

```
(D, 0, 3; A, 0, 1)  (D, 0, 4; A, 0, 1)  (S, 0, 3; A, 0, 1)
(S, 0, 4; A, 0, 1)  (S, 0, 4; B, 3, 4)  (S, 1, 4; B, 3, 4)
```

The combine phase adds the single proposal

```
(D, 0, 4; B, 3, 4)
```

after which the system is stable.

14.4.3.1 Time Requirements of the Rytter Recognizer

The reader may by now be wondering what is gained by all this. After all, if we compare this to CYK, we only seem to do extra work. A CYK derivation step that combines (A,i,j) with (B,j,k) to (C,i,k) if we have a rule $C \rightarrow AB$ seems to have been split in two: (A,i,j) leads to $(C,i,k;B,j,k)$ (from the propose phase), and $(C,i,k;B,j,k)$ and (B,j,k) together lead to (C,i,k) (from the recognition phase). Indeed, without the combining of Rytter proposals we would not have gained anything. However, from our example above we learned that after one iteration of the algorithm, there already are realized CYK hypotheses of length 4. In fact, the sentence has been recognized after a single iteration. The algorithm is so powerful that sentences of length 2^n can be recognized in $O(n)$ iterations, in other words, sentences of length n can be recognized in $O(^2 \log\ n)$ iterations. To see why this is so, we first need to introduce some terminology.

We define the *size of a CYK hypothesis* (A,i,j) as $(j-i)$, the length of the substring covered by the hypothesis. We also define the *size of a Rytter proposal* $(A,i,j;B,o,p)$ as $size(A,i,j) - size(B,o,p)$, which is the length of the covered substring minus the length of the gap. So, the size is the length of the recognized part.

In the remainder of this section, we will show that

I_k If (A,i,j) is a realizable CYK hypothesis with size $\leq 2^k$, it will be realized (be a member of S_{CYK}) after at most k iterations of the algorithm.

II_k If $(A,i,j;B,o,p)$ is a realizable Rytter proposal with size $\leq 2^{k-1}$, it will be realized (be a member of S_{Rytter}) after at most k iterations of the algorithm.

We will do so by showing that II_k and I_k, together, imply II_{k+1} and I_{k+1}.

For a complete proof, we also need I_0 and II_0 but these are trivial.

First, we will show that II_k and I_k imply II_{k+1}. Suppose that $(A,i,j;B,o,p)$ is a realizable Rytter proposal with $2^{k-1} < size(A,i,j;B,o,p) \leq 2^k$. We then have to show that it is a member of S_{Rytter} after $k+1$ iterations. Since $(A,i,j;B,o,p)$ is realizable, there exists a (partial) parse tree with root A for the substring $s_{i+1} \cdots s_j$, and somewhere in this tree there is a leaf node B which is supposed to fill the gap. As there is a path from A to every leaf in this tree, there is also a path to node B. The list of non-terminals on this path is denoted C, with $C_0 = A, \ldots, C_l = B$, and each C_q on this path derives a substring $s_{m_q+1} \cdots s_{n_q}$ that contains the substring derived from C_{q+1}: $s_{m_{q+1}+1} \cdots s_{n_{q+1}}$ (all with a gap for non-terminal B). This means that all Rytter proposals $(C_r,m_r,n_r;C_q,m_q,n_q)$ with $0 \leq r < q \leq l$ are realizable. Such a list of CYK hypotheses (C_q,m_q,n_q) is called a *hypothesis path* for the realizable Rytter proposal $(A,i,j;B,o,p)$. See Figure 14.14, where we only show the non-terminals in the path.

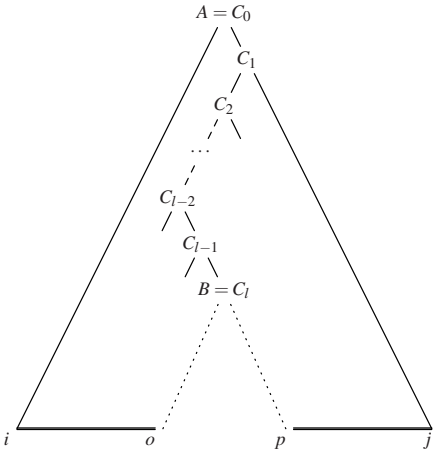


Fig. 14.14. A hypothesis path for the realizable Rytter proposal $(A,i,j;B,o,p)$

A step from r to $r-1$ in this hypothesis path is called a *critical step* if $size(A,i,j;C_{r-1},m_{r-1},n_{r-1}) \leq 2^{k-1}$ and $size(C_r,m_r,n_r;B,o,p) \leq 2^{k-1}$. We will now first show that any hypothesis path of every realizable Rytter proposal has a critical step. The size of the Rytter proposals $(A,i,j;C_q,m_q,n_q)$ is increasing monotonically with q, because the gap is shrinking. In the limits, $size(A,i,j;C_0,m_0,n_0) = size(A,i,j;A,i,j) = 0$, and $2^(k-1) < size(A,i,j;B,o,p) = size(A,i,j;C_l,m_l,n_l) \leq 2^k$. This means that there exists an r such that $size(A,i,j;C_{r-1},m_{r-1},n_{r-1}) \leq 2^{k-1}$ and $2^{k-1} < size(A,i,j;C_r,m_r,n_r) \leq 2^k$. Now, we have the following:

- since $2^{k-1} < size(A,i,j;C_r,m_r,n_r) \leq 2^k$, this means that $2^{k-1} < (j-i) - (n_r - m_r) \leq 2^k$;
- since $2^{k-1} < size(A,i,j;B,o,p) \leq 2^k$, this means that $2^{k-1} < (j-i) - (p-o) \leq 2^k$.
- $n_r - m_r \geq p - o$.

So, $2^{k-1} < (j-i) - (n_r - m_r) \leq (j-i) - (p-o) \leq 2^k$, and thus $2^{k-1} < (j-i) - (n_r - m_r) \leq (j-i) - (n_r - m_r) + (n_r - m_r) - (p-o) \leq 2^k$. This also means that $2^{k-1} + (n_r - m_r) - (p-o) \leq 2^k$, so that $(n_r - m_r) - (p-o) \leq 2^k - 2^{k-1} = 2 \times 2^{k-1} - 2^{k-1} = 2^{k-1}$. This proves that $size(C_r,m_r,n_r;B,o,p) = (n_r - m_r) - (p-o) \leq 2^{k-1}$. This means that the step from r to $r-1$ is a critical step.

Now suppose r to $r-1$ is a critical step in the hypothesis path (C_q,m_q,n_q) for the realizable Rytter proposal $(A,i,j;B,o,p)$. Then there is either a realizable CYK hypothesis (D,m_{r-1},m_r) that combines with (C_r,m_r,n_r) to form (C_{r-1},m_{r-1},n_r) (in which case $n_{r-1} = n_r$ and there is a grammar rule $C_{r-1} \rightarrow DC_r$), or there is a realizable CYK hypothesis (D,n_r,n_{r-1}) that combines with (C_r,m_r,n_r) to form (C_{r-1},m_r,n_{r-1}) (in which case $m_{r-1} = m_r$ and there is a grammar rule $C_{r-1} \rightarrow C_rD$). In either case, we call the CYK hypothesis with non-terminal D hyp_D.

Figure 14.15 illustrates a hypothesis path with its critical step; some consequences are shown to the right of the figure. Since D represents the filled

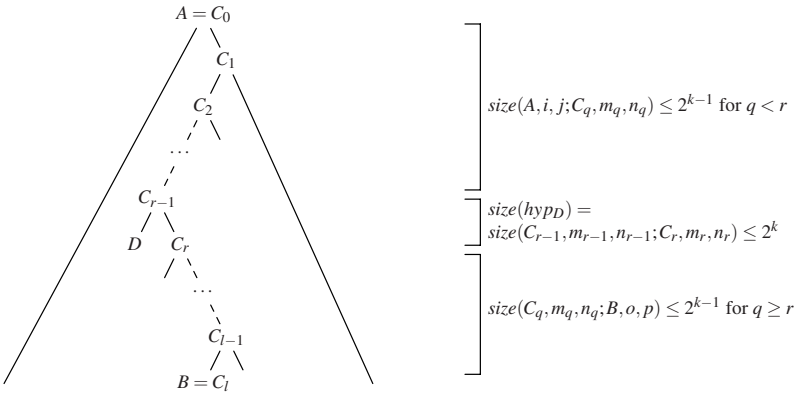


Fig. 14.15. A critical step in a hypothesis path

part in the Rytter proposal $(C_{r-1},m_{r-1},n_{r-1};C_r,m_r,n_r)$, we have $size(hyp_D) = size(C_{r-1},m_{r-1},n_{r-1};C_r,m_r,n_r)$, so that $size(hyp_D) \leq 2^k$. Therefore, we know from I_k that hyp_D is a member of S_{CYK} after k iterations. Also, from II_k we know that both $(A,i,j;C_{r-1},m_{r-1},n_{r-1})$ and $(C_r,m_r,n_r;B,o,p)$ are a member of S_{Rytter} after k iterations.

Now let us consider the steps of iteration $k+1$. The presence of hyp_D in S_{CYK} and $(C_r,m_r,n_r;B,o,p)$ in S_{Rytter} prompts the first part of the combine phase to propose, among others, the Rytter proposal $(C_{r-1},m_{r-1},n_{r-1};B,o,p)$. Next, the second stage of the combine phase of this iteration kicks in and combines this proposal with the proposal $(A,i,j;C_{r-1},m_{r-1},n_{r-1})$, which was already a member of S_{Rytter}, to produce the Rytter proposal $(A,i,j;B,o,p)$. This proves that II_k and I_k imply II_{k+1}.

Now, we have to show that I_k and II_{k+1} imply I_{k+1}. Again, we look more closely at how a CYK hypothesis (A,i,j), with $2^{k-1} < size(A,i,j) \leq 2^k$ ends up in S_{CYK}: except for the initialization hypotheses, it must have come from a recognize step: some Rytter proposal $(A,i,j;B,o,p)$ must be a member of S_{Rytter}, and (B,o,p) a member of S_{CYK}. The hypothesis (B,o,p) is called a *critical hypothesis* for (A,i,j) if $2^{k-1} < size(B,o,p) \leq 2^k$, and there is a grammar rule $B \rightarrow CE$ and there are realizable CYK hypotheses (C,o,l) and (E,l,p), both with size less than or equal to 2^{k-1}. Of course, (A,i,j) can also be its own critical hypothesis. In Problem 14.7 the reader is invited to prove that every realizable CYK hypothesis with size ≥ 2 has a critical hypothesis. Also, if $(A,i,j;B,o,p)$ is a realizable Rytter proposal, and (B,o,p) is a critical hypothesis to (A,i,j), and $2^{k-1} < size(A,i,j) \leq 2^k$, then $size(A,i,j;B,o,p) \leq 2^{k-1}$.

Now, suppose (A,i,j) is a realizable CYK hypothesis with $2^k < size(A,i,j) \leq 2^{k+1}$, and (B,o,p) is a critical hypothesis for (A,i,j). This means that there exist two realizable CYK hypotheses (C,o,l) and (E,l,p), each of size $\leq 2^k$ and there is a grammar rule $B \rightarrow CE$.

Let us first assume that $(A,i,j) \neq (B,o,p)$. $size(A,i,j;B,o,p) \leq 2^k$. As we have seen above, this means that it gets proposed at the latest in the combine phase of iteration $k+1$.

Since S_{CYK} contains (C,o,l) and (E,l,p), the CYK combination in the combine phase of iteration $k+1$ adds (B,o,p) to S_{CYK}. Next, the recognize phase of the same iteration combines this with $(A,i,j;B,o,p)$ to produce (A,i,j).

The case that $(A,i,j) = (B,o,p)$ is left as an exercise to the reader, as is the case that (A,i,j) did not end up in S_{CYK} as the result of a recognize step combining a Rytter proposal with a CYK hypothesis, but was the result of combining two CYK hypotheses.

Since Rytter [226, 227] does not combine CYK hypotheses, and also does not combine Rytter proposals with CYK hypotheses to produce new Rytter proposals, Rytter's version of the algorithm actually needs more than $^2\log n$, but still $O(\log n)$ iterations. Sikkel [158] proves that if you change the order of the steps in the iteration to "recognize, propose, combine, combine" (do the combine twice), the algorithm needs at most $^2\log n$ iterations. This order, however, seems less intuitive.

All in all, we now have seen that the algorithm needs $O(\log(n))$ iterations to determine if $(S_S,0,n)$ is a member of S_{CYK}. What remains to be determined is that each iteration can be executed in constant time, given enough processors. In the next section, we will show how to build a Boolean circuit that will do just that.

14.4.3.2 A Rytter Recognizer on a Boolean Circuit

We will now examine how each phase of the Rytter algorithm translates into a part of a Boolean circuit, but first we need to clear up some notation. Instead of using $A_{i,l}$, as we did earlier for denoting a node that represents the hypothesis that non-terminal A recognizes a string of length l, starting at index i, we now use the indices as used in the description of the Rytter algorithm, so the notation would be $A_{i-1,i+l-1}$ for the same substring.

In the following, we switch to Rytter's description of the algorithm, because it is a bit simpler: it does not have an explicit combine of CYK hypotheses, and no combination of Rytter proposals and CYK hypotheses to produce a new Rytter proposal.

As in the CYK circuit, for each non-terminal and substring $s_{i+1}\cdots s_j$ we have an OR-node $A_{i,j}$. We also have OR-nodes for all possible Rytter proposals $(A,i,j;B,o,p)$. We will denote these nodes as $A_{i,j}\backslash B_{o,p}$.

Let us first look at the inputs of a node $A_{i,j}$. We know that the CYK hypothesis (A,i,j) is realizable if a Rytter proposal $(A,i,j;B,o,p)$ is realizable and (B,o,p) is realizable as well. So, an input of $A_{i,j}$ is the output of an AND-node with inputs from both $A_{i,j}\backslash B_{o,p}$ and $B_{o,p}$. This means that we have AND-nodes for each Rytter proposal $A_{i,j}\backslash B_{o,p}$ and CYK hypothesis $B_{o,p}$. This completes the recognize phase of the Rytter algorithm.

Now we turn our attention to the inputs of a node $A_{i,j}\backslash B_{o,p}$. A Rytter proposal can be realized in two ways: the proposal phase of the Rytter algorithm can propose it, or it can be constructed in the combine phase. Rytter proposals that are the result of the proposal phase have either the form $(A,i,j;B,p,j)$ or $(A,i,j;B,i,o)$. In the

first case, there is a grammar rule $A \rightarrow CB$ and the "origin" of the proposal is a CYK hypothesis (C,i,p), so the output of $C_{i,p}$ must be an input of $A_{i,j}\backslash B_{p,j}$. In the second case, there is a grammar rule $A \rightarrow BC$ and the "origin" of the proposal is a CYK hypothesis (C,o,j), so the output of $C_{o,j}$ must be an input of $A_{i,j}\backslash B_{i,o}$.

The combine phase combines two Rytter proposals to one of a larger size. The outputs of the two Rytter proposals go to an AND-node, whose output serves as input for the result proposal: if both $A_{i,j}\backslash B_{o,p}$ and $B_{o,p}\backslash C_{k,l}$ are "on", both inputs of an AND-node, tentatively named $A_{i,j}\backslash B_{o,p}\backslash C_{k,l}$ are "on", so its output will be "on" in the next time step, turning node $A_{i,j}\backslash C_{k,l}$ on in the time step after that. We will call such a node a *Rytter combine node*.

This completes the building blocks of the Rytter recognizing circuit. As was the case with the CYK recognizing circuit, when the circuit is stabilized, the output of node $S_{0,n}$ determines whether a sentence $s_1 \cdots s_n$ has been recognized or not.

We will not work out a practical example here. The next section will show the reader why not.

14.4.3.3 Node Requirements for the Rytter Recognizer Circuit

In the previous section, we have seen the kinds of nodes that are needed for the Rytter recognizer circuit. There are

- OR-nodes for each CYK hypothesis (A,i,j), so for each non-terminal A and each i and j such that $0 \leq i < j \leq n$. The number of these nodes is $O(n^2)$.
- OR-nodes for each Rytter proposal $(A,i,j;B,o,p)$, so for all non-terminals A and B and each i, j, o, and p, such that $0 \leq i \leq o < p \leq j \leq n$. So, there are $O(|V_N|^2 n^4)$ such nodes, where V_N is the set of non-terminals.
- AND-nodes that we tentatively named $A_{i,j}\backslash B_{o,p}\backslash C_{k,l}$, the Rytter combine nodes. There are many of those, for each non-terminal A, B, C, and i,j,o,p,k,l such that $0 \leq i \leq o \leq k < l \leq p \leq j \leq n$, so as many as $|V_N|^3 n^6$ nodes.
- AND-nodes for the recognize phase, there is one for each Rytter proposal.
- and finally, the nodes that are needed to initialize the circuit, exactly as with the CYK recognizing circuit. This are $O(n)$ nodes.

So, the total number of nodes is dominated by the AND-nodes that are needed for the combine phase, and thus the total number of nodes is $O(n^6|V_N|^3)$. Again, although the time requirement is independent of $|V_N|$, the number of nodes needed depends on it, even more heavily than in the simple CYK recognizer.

The number of nodes required is considerable. Even a very small example, with an input of length 3, and a grammar like the one from Figure 14.9, with 4 non-terminals, would need the following nodes (where $C(k,n)$ is the number of combinations of $i_1 \ldots i_k$ that fulfill $0 \leq i_1 \leq \ldots \leq i_k \leq n$): $4 \times C(2,3)$ OR-nodes for the CYK hypotheses, $+\ 4^2 \times C(4,3)$ OR-nodes for the Rytter proposals, $+\ 4^3 \times C(6,3)$ AND-nodes for the combination phase, $+\ 4^2 \times C(4,3)$ AND-nodes for the recognition phase, $+\ 4^1$ initialization nodes $= 4 \times 10 + 16 \times 35 + 64 \times 84 + 16 \times 35 + 4 = 6540$ nodes. So a detailed picture like the one in Figure 14.13 is out of the question, and it would not be very interesting for the reader! Note, however, that it is less than 1/7

of the number suggested by the order-of-magnitude formula above: $4^3 \times 3^6 = 46656$ nodes. An Application-Specific Integrated Circuit (ASIC) for parsing a sentence of length 10 with a grammar with 10 non-terminals would have $O(10^9)$ gates.

The data structures of the Rytter algorithm show interesting similarities to those used in the parser for tree-adjoining grammars in Section 15.4.2.

14.4.3.4 Turning the Rytter Recognizer into a Parser

When the recognizer algorithm is finished (no more items are added during a complete iteration of the algorithm), the set of CYK hypotheses generally contains many hypotheses that did not contribute to the recognizing of the sentence. So, let us filter out the CYK hypotheses that should be part of the parse forest. These hypotheses are particularly easy to find: they are exactly the members (A,i,j) of S_{CYK} for which $(S,0,n;A,i,j)$ is a realized Rytter proposal. On the one hand, if both $(S,0,n;A,i,j)$ and (A,i,j) are realized, $(S,0,n)$ will be realized, so the realizability of (A,i,j) contributes to the recognition of the input. On the other hand, if (A,i,j) contributes to the recognition of the input, this means two things:

- non-terminal A derives substring $s_{i+1}\cdots s_j$, so (A,i,j) will be realized at some stage of the algorithm;
- S derives the sentential form $s_1\cdots s_i A s_{j+1}\cdots s_n$, so $(S,0,n;A,i,j)$ will be realized at some stage of the algorithm.

So, this gives us all nodes in the parse forest, which is, strictly speaking, together with the grammar rules, enough to build the actual parse forest.

In our Boolean circuit for sentences of length n, we add an AND-node $A'_{i,j}$ for each node $A_{i,j}$, with two inputs: one from node $A_{i,j}$ and one from $S_{0,n}\backslash A_{i,j}$. Perhaps amazingly, it only takes one time-step extra (over just recognizing) to determine all CYK hypotheses that are part of the parse forest. If this result is not quite satisfactory, for instance because it does not represent a "real" parse forest, the reader is referred to exercise 14.8.

14.5 Conclusion

We have discussed three methods of parallelizing the parsing process: multiple serial parsers, process-configuration parsers, and connectionist parsers. We have also seen examples of each. The boundaries between the three parallelizing methods are not always that clear, however; the difference between process-configuration and connectionist is more a matter of scale than anything else. The CYK parser, and even the Rytter parser, both presented here as connectionist parsers, could also be considered process-configuration parsers, albeit with extremely simple agents, and a large number of them. The literature references in (Web)Section 18.2.5 contain more examples, as do the references in the parallel parsing bibliography by Alblas et al. [234].

The parallelizing techniques presented here are quite parse-method specific. Janssen et al. [232] and Sikkel [158] describe the "primordial soup algorithm", a

mechanism that allows for the specification of various parallel parsing algorithms without specifying flow control or data structures. This gives an abstract, elegant, and compact mathematical basis for the design of a parallel implementation.

Problems

Problem 14.1: What should a Fischer parser do when it accepts its part as a complete sentence (in the example of Section 14.2: reduces its part to `S`)?

Problem 14.2: Apply the Fischer parser on the LR(0) example of Figure 9.18, on two processors, the sentence `n-(n-n)-(n)$`, using two different splits: 1. `n-(n-n` and `)-(n)$` (equal length split), and 2. `n-(n-n)` and `-(n)$`. Comment.

Problem 14.3: Read the paper by Bar-On and Vishkin [225] and write a program implementing their algorithm.

Problem 14.4: *Project:* Analyze the performance of the Sikkel and Lankhorst algorithm of Section 14.3.1. In your analysis, also consider the dependency on the size of the grammar.

Problem 14.5: Refer to Section 14.4.2.2. Explain why metaparsing and a reversed scan does not in general result in a circuit that represents a parse forest.

Problem 14.6: Proving something by calling it trivial is sometimes called "proof by intimidation". [2] In Section 14.4.3.1 we called I_0 and II_0 trivial. The reader is invited to check.

Problem 14.7: Refer to Section 14.4.3.1. Why does every realizable CYK hypothesis (A,i,j) of size 2^k with $k \geq 1$ always have a critical hypothesis?

Problem 14.8: Refer to Section 14.4.3.4, where a Boolean circuit is discussed that determines the CYK hypotheses that are actually nodes in the parse forest. Show how to extend this circuit so that it also reflects which grammar rules are used, along the lines of Figure 14.13. However, the circuit must still stabilize in $O(^2\log n)$ time steps.

[2] A math professor did this while teaching, and was asked why the proof was trivial. He then left the room, to come back 15 minutes later and say: "Indeed, it is trivial", and then proceeded, without further clarification.

15

Non-Chomsky Grammars and Their Parsers

Just as the existence of non-stick pans points to user dissatisfaction with "sticky" pans, the existence of non-Chomsky grammars points to user dissatisfaction with the traditional Chomsky hierarchy. In both cases ease of use is the issue.

As we have seen in Section 2.3, the Chomsky hierarchy consists of five levels:

- phrase structure (PS),
- context-sensitive (CS),
- context-free (CF),
- regular (finite-state, FS) and
- finite-choice (FC).

Although each of the boundaries between the types is clear-cut, some boundaries are more important than others. Two boundaries specifically stand out: that between context-sensitive and context-free and that between regular (finite-state) and finite-choice. The significance of the latter is trivial, being the difference between productive and non-productive, but the former is profound.

The border between CS and CF is that between global correlation and local independence. Once a non-terminal has been produced in a sentential form in a CF grammar, its further development is independent of the rest of the sentential form, but a non-terminal in a sentential form of a CS grammar has to look at its neighbors on the left and on the right, to see what production rules are allowed for it. The local production independence in CF grammars means that certain long-range correlations cannot be expressed by them. Such correlations are, however, often very interesting, since they embody fundamental properties of the input text, like the consistent use of variables in a program or the recurrence of a theme in a musical composition.

15.1 The Unsuitability of Context-Sensitive Grammars

The obvious approach would be using a CS grammar to express the correlations (= the context-sensitivity) but here we find our way obstructed by three practical

rather than fundamental problem areas: understandability, parsability, and semantic suitability.

15.1.1 Understanding Context-Sensitive Grammars

CS grammars *can* express the proper correlations but not in a way a human can understand. It is in this respect instructive to compare the CF grammars in Section 2.3.2 to the one CS grammar we have seen that really expresses a context-dependency, the grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ from Figure 2.7, repeated here in Figure 15.1. The grammar for

```
 Ss  →  abc | aSQ
bQc  →  bbcc
 cQ  →  Qc
```

Fig. 15.1. Context-sensitive grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

the contents of a book (Figure 2.10) immediately suggests the form of the book, but the grammar of Figure 15.1 hardly suggests anything, even if we can still remember how it was constructed and how it works. This is not caused by the use of short names like `Q`: a version with more informative names (Figure 15.2) is still puzzling.

```
           Ss  →  a b c | a S bc_pack
b bc_pack c    →  b b c c
  c bc_pack    →  bc_pack c
```

Fig. 15.2. Context-sensitive grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ with more informative names

Also, one would expect that, having constructed a grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, making one for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n$ would be straightforward. That is not the case; a grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n$ requires rethinking of the problem (see Problem 15.1).

The cause of this misery is that CS and PS grammars derive their power to enforce global relationships from "just slightly more than local dependency". Theoretically, just looking at the neighbors can be proved to be enough to express any global relation, but the enforcement of a long-range relation through this mechanism causes information to flow through the sentential form over long distances. In the production process of, for example $\mathtt{a}^4\mathtt{b}^4\mathtt{c}^4$, we see several `bc_pack`s wind their way through the sentential form, and in any serious CS grammar, many messengers run up and down the sentential form to convey information about developments in far-away places. However interesting this imagery may be, it requires almost all rules to know something about almost all other rules; this makes the grammar absurdly complex.

15.1.2 Parsing with Context-Sensitive Grammars

Parsing speed is also an issue. FS parsing can always be done in linear time; many CF grammars automatically lead to linear-time parsers; CF parsing needs never be more expensive than $O(n^3)$ and is usually much better; but no efficient parsing algorithms for CS or PS grammars are known. CS parsing is basically exponential in general — although (Web)Section 18.1.1 reports some remarkable efforts — and PS parsing is not even solvable in theory.

Still, many CS languages can be recognized in linear time, using standard CF parsing techniques almost exclusively. The language $a^n b^n c^n$ is a good example. We have already observed (at the beginning of Chapter 13) that it is the intersection of two CF languages, $a^n b^n c^m$ and $a^m b^n c^n$: the first language forces the numbers of **a**s and **b**s to be equal, the second does the same for the **b**s and the **c**s. We can easily create linear-time recognizers for both languages, for example by writing LL(1) grammars for them. We can then test the string with both recognizers, and if they both recognize the string, it belongs to the language $a^n b^n c^n$; otherwise it is rejected. This test can be done in linear time. So it is not totally unreasonable to demand good recognition speed for at least some non-CF languages. Constructing a parse tree depends on the exact form of the grammar and may be much more expensive; the result may not even be a tree but a dag, as it was in Figure 2.8.

15.1.3 Expressing Semantics in Context-Sensitive Grammars

A third problem concerns the semantic suitability. Although this book has not emphasized the semantic aspect of language processing (see Section 2.11), that aspect is of course important for anybody who is interested in the results of a parsing. In FS systems, semantic actions can be attached to regular expressions or transitions (see Section 5.9). CF grammars are very convenient for expressing semantics: to each production rule $A \rightarrow A_1 A_2 \cdots A_k$ code can be attached that composes the semantics of A from that of its children $A_1, A_2, \ldots, A_k$. But it is less than clear where we can find or attach semantics in a CS rule like `b bc_pack c -> b b c c`.

15.1.4 Error Handling in Context-Sensitive Grammars

Less important than the above but still an issue in practice is the behavior of a parsing technique on incorrect input: error detection ("Is there an error?"), error reporting ("Where exactly is the error and what is it?") and error repair ("Can we repair and continue?"). As we shall see in Chapter 16, error handling with CF grammars is a difficult area in which only moderately good answers are known. Error handling with *non-CF* grammars can be a nightmare. Already error *detection* can be a serious problem, since the parser is easily tempted to try an infinite number of increasingly complex hypotheses to explain the unexplainable: incorrect input then leads to nontermination. And given a non-Chomsky parser for the language ww, where w is an arbitrary string of **a**s and **b**s, and the input **aabbaabbab**, where exactly is the error, and what would be a sensible error message?

15.1.5 Alternatives

Many grammar forms have been put forward to mitigate the above problems and make long-range relationships more easily expressible. We shall look at several of them, with a special eye to understandability, parsability, semantic suitability, and error handling. More in particular we will look at VW grammars, attribute grammars, affix grammars, tree-adjoining grammars, coupled grammars, ordered grammars, recognition systems, Boolean grammars, and §-calculus. Of these, the recognition systems and to a certain extent §-calculus are particularly interesting since they question the wisdom of describing sets by generative means at all. Given the large variety of non-Chomsky systems, the relative immaturity of most of them, and the limited space, our descriptions of these systems will be shorter than those in the rest of this book. The bibliography in (Web)Section 18.2.6 contains explanations of several other non-Chomsky systems.

One interesting possibility not explored here is to modify the CF grammar under the influence of parser actions. This leads to *dynamic grammars*, also called *modifiable grammars*, or *adaptable grammars*. As the names show, the field is still in flux. See Rußmann [280] for theory and practice of LL(1) parsing of dynamic grammars. (Web)Section 18.2.6 contains many more references on the subject.

Each of the non-Chomsky systems should come with a paradigm, telling the user how to look at his problem so as to profit best from it. For a CF grammar this paradigm is fairly obvious, but even for a CS grammar it is not, as the grammar of Figure 15.1 amply shows. With the exception of VW grammars and attribute grammars, not enough experience has been gathered to date with any of the non-Chomsky systems for a paradigm to emerge. Also, VW grammars and attribute grammars are the only ones of the methods discussed here that can more or less conveniently describe large real-world context-sensitive systems, as the ALGOL 68 report [244] and several compilers based on attribute grammars attest.

There is one example of a non-Chomsky grammar type for CF languages: Floyd productions; they were already discussed in Section 9.3.2. Push-down automata (Section 6.2) could be considered another.

15.2 Two-Level Grammars

It is not quite true that CF grammars cannot express long-range relations; they can only express a finite number of them. If we have a language the strings of which consist of a **`begin`**, a **`middle`** and an **`end`** and suppose there are three types of **`begin`**s and **`end`**s, then the CF grammar of Figure 15.3 will enforce that the type of the **`end`** will properly match that of the **`begin`**, independent of the length of **`middle`**.

We can think of **`(`** and **`)`** for **`begin1`** and **`end1`**, **`[`** and **`]`** for **`begin2`** and **`end2`** and **`{`** and **`}`** for **`begin3`** and **`end3`**; the CF grammar will then ensure that each closing parenthesis will match the corresponding open parenthesis.

```
text_S  ->  begin1 middle end1
        |   begin2 middle end2
        |   begin3 middle end3
```

Fig. 15.3. A long-range relation-enforcing CF grammar

By making the CF grammar larger and larger, we can express more and more long-range relations; if we make it infinitely large, we can express any number of long-range relations and have achieved full context-sensitivity. Now we come to the fundamental idea behind two-level grammars. The rules of the infinite-size CF grammar form an infinite set of strings; in other words, it is a language, which can in turn be described by a grammar. This explains the name *two-level grammar*.

The type of two-level grammar described in this section was invented by van Wijngaarden [244] and is often called a *VW grammar*. There are other kinds of two-level grammars, for example those by Krulee [270]; and some ordered grammars (Section 15.6) also use two grammars.

15.2.1 VW Grammars

To introduce the concepts and techniques we shall give here an informal construction of a VW grammar for the language $L = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ for $n \geq 1$ from the previous section. We shall use the VW notation as explained in Section 2.3.2.3: the names of terminal symbols end in **`symbol`** and their representations are given separately; rules are terminated by a dot (**`.`**); alternatives are separated by semicolons (**`;`**); members inside alternatives are separated by commas, allowing us to have spaces in the names of non-terminals; and a colon (**`:`**) is used instead of an arrow to separate left- and right-hand side.

Using this notation, we could describe the language L through a context-free grammar if grammars of infinite size were allowed:

```
text_S:  a symbol, b symbol, c symbol;
         a symbol, a symbol,
               b symbol, b symbol,
               c symbol, c symbol;
         a symbol, a symbol, a symbol,
               b symbol, b symbol, b symbol,
               c symbol, c symbol, c symbol;
         ...       ...
```

We shall now try to master this infinity by constructing a grammar which allows us to produce the above grammar as far as needed. We first introduce an infinite number of names of non-terminals:

```
text_S:  ai, bi, ci;
         aii, bii, cii;
         aiii, biii, ciii;
         ...       ...
```

together with three infinite groups of rules for these non-terminals:

```
ai:    a symbol.
aii:   a symbol, ai.
aiii:  a symbol, aii.
...    ...

bi:    b symbol.
bii:   b symbol, bi.
biii:  b symbol, bii.
...    ...

ci:    c symbol.
cii:   c symbol, ci.
ciii:  c symbol, cii.
...    ...
```

Here the **i** characters count the number of **a**s, **b**s and **c**s. Next we introduce a special kind of name called a *metanotion*. Rather than being capable of producing (part of) a sentence in the language, it is capable of producing (part of) a name in a grammar rule. In our example we want to catch the repetitions of **i**s in a metanotion **N**, for which we give a context-free production rule (a *metarule*):

```
N :: i ; i N .
```

Note that we use a slightly different notation for metarules: left-hand side and right-hand side are separated by a double colon (**::**) rather than by a single colon and members are separated by a blank ( ) rather than by a comma; also, the metanotion names consist of upper case letters only (but see the note on numbered metanotions in Section 15.2.2). The set of metarules in a VW grammar is called the *metagrammar*. The metanotion **N** produces the segments **i**, **ii**, **iii**, etc., which are exactly the parts of the non-terminal names we need.

We can use the production rules of **N** to collapse the four infinite groups of rules into four *finite* rule templates called *hyperrules*, as shown in Figure 15.4.

```
N ::    i ; i N .

text_S:  a N, b N, c N.

a i:    a symbol.
a i N:  a symbol, a N.

b i:    b symbol.
b i N:  b symbol, b N.

c i:    c symbol.
c i N:  c symbol, c N.
```

Fig. 15.4. A VW grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

Each original rule can be obtained from one of the hyperrules by substituting a production of **N** from the metarules for each occurrence of **N** in that hyperrule, provided that *the same production* of **N** is used consistently throughout; this form of substitution is called *consistent substitution*. To distinguish them from normal names, these half-finished combinations of lower case letters and metanotions (like **a N** or **b i N**) are called *hypernotions*. Substituting, for example, **N=iii** in the hyperrule

```
b i N: b symbol, b N.
```

yields the CF rule for the CF non-terminal **biiii**:

```
biiii: b symbol, biii.
```

We can also use this technique to condense the *finite* parts of a grammar by having a metarule **A** for the symbols **a**, **b** and **c**. ("**A**" stands for "alphabetic".) Again the rules of the game require that the metanotion **A** be replaced consistently. The final result is shown in Figure 15.5. We see that even the names of the terminal symbols

```
N ::    i ; i N .
A ::    a ; b ; c .

text_S:  a N, b N, c N.
A i:     A symbol.
A i N:   A symbol, A N.
```

Fig. 15.5. The final VW grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

are generated by the grammar; this feature is exploited further in Section 15.2.5.

This grammar gives a clear indication of the language it describes: once the "value" of the metanotion **N** is chosen, production is straightforward. It is now trivial to extend the grammar to $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n\mathbf{d}^n$. It is also clear how long-range relations are established without having confusing messengers in the sentential form: they are established *before* they become long-range, through consistent substitution of metanotions in simple right-hand sides. The "consistent substitution rule" for metanotions is essential to the two-level mechanism; without it, VW grammars would be equivalent to CF grammars (Meersman and Rozenberg [253]).

A very good and detailed explanation of VW grammars has been written by Cleaveland and Uzgalis [252], who also show many applications. Sintzoff [241] has proved that VW grammars are as powerful as PS grammars, which also shows that adding a third level to the building cannot increase its powers. van Wijngaarden [249] has shown that the metagrammar need only be regular (although simpler grammars may be possible if it is allowed to be CF).

When the metagrammar is restricted to a finite-choice grammar, that is, each metanotion just generates a finite list of words, the generation of the CF grammar rules from the hyperrules can be performed completely, and the result is a (much larger) set of CF rules. Conversely, the use of a finite metalevel can often reduce considerably the number of rules in a grammar; we used this in the grammar of

Figure 15.5 when we condensed the **a**, **b** and **c** into a metanotion **A**. In linguistics, the attributes of words are very often finite:

```
GENDER ::  masculine ; feminine ; neuter .
NUMBER ::  singular ; plural .
MODE ::    indicative ; subjunctive ; optative .
...
```

and using them as (finite-choice) metanotions can help reduce the complexity of the grammar.

15.2.2 Expressing Semantics in a VW Grammar

Now that we have seen that the understandability of VW grammars is excellent, we will turn to the semantic suitability. Here we are in for a pleasant surprise: van Wijngaarden [257] has shown that VW grammars can produce the semantics of a program *together with* that program. In short, we do not have to leave the formalism to express the semantics.

For an almost trivial example, let us assume the semantics of a string $\mathbf{a}^i\mathbf{b}^i\mathbf{c}^i$ is the string "**OK**" if $i > 5$ and "**KO**" otherwise. The grammar in Figure 15.6 then produces strings of the form

```
aaa...bbb...ccc...=>[OK|KO]
```

with the proper "**OK**" or "**KO**".

```
1. N ::                  i ; i N .
2. A ::                  a ; b ; c .

3. text_S :              a N, b N, c N,
                             result symbol, semantics N.
4. A i:                  A symbol.
5. A i N:                A symbol, A N.

6. semantics iiiii N:    ok symbol.
7. semantics N:          where N N1 equals iiiiii, ko symbol.
8. where N equals N:     .
```

Fig. 15.6. VW grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ => [**OK**|**KO**]

Rule number 6 in the grammar of Figure 15.6 says that if the original **N** can be split up in five **i**s (the **iiiii**) and a sequence of at least one more **i** (the **N**), then the semantics is "**OK**". Rule number 7 uses a so-called *predicate*, a rule that controls the production process by either producing nothing (success) or getting stuck (failure). The hypernotion **where N N1 equals iiiiii** succeeds only when **N1** can be chosen so that **N N1** forms **iiiiii**, that is, when there is a number **N1** larger than zero that can be added to **N** to form 6, that is, when **N** is less than 6. Any sentential form that includes a notion like **where i equals ii** is a blind alley since

no hyperrule will produce a CF rule with **where i equals ii** as its left-hand side. The methods and techniques used here belong to the two-level programming paradigm; the ALGOL 68 report [244] is full of them.

The above paragraph uses another standard feature of VW grammars, the creation of independent copies of a metanotion by appending a number to its name. The **N1** in the above hypernotion is an independent copy of the metanotion **N** and is different from the **N** that occurs in the same rule. All **N**s must be substituted consistently, and so must all **N1**s, etc., as far as applicable.

If we consider the VW grammar to be a grammar for a programming language, the above technique produces sentences consisting of programs (sequences $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ in the above example) with their semantics. We can carry the semantic expression process one step further, leave out the program at all and just produce the result from a formulation of the problem in a VW grammar. The following small VW grammar produces the result of the multiplication **N1×N2**, given the above definition of **N** (note the similarity to definitions from mathematics and functional programming):

```
produce N1 times N2 i: produce N1 times N2, write N1.
produce N1 times i: write N1.
write N i: write N, i symbol.
write i: i symbol.
```

Given the start symbol **produce iii times iiii**, this grammar produces one string: **iiiiiiiiiiii**. So rather than having a grammar that we use to produce a program that we run to obtain a result, we have a grammar that we run to obtain a result. This explains the title *Languageless programming* of the paper in which van Wijngaarden [257] describes this technique. Małuszyński [261] develops the idea further. Grune [260] describes a sentence producing program for VW grammars; the above examples run correctly on this program. The first 8 lines of the output for the grammar from Figure 15.6 are given in Figure 15.7.

```
abc=>KO.
aabbcc=>KO.
aaabbbccc=>KO.
aaaabbbbcccc=>KO.
aaaaabbbbbccccc=>KO.
aaaaaabbbbbbcccccc=>OK.
aaaaaaabbbbbbbcccccccc=>OK.
aaaaaaaabbbbbbbbcccccccc=>OK.
```

Fig. 15.7. The first 8 lines of output for the grammar from Figure 15.6

These examples are almost trivial, but they do show an outline of what can be done: it is a paradigm in its infancy. If VW programming ever becomes a full-fledged paradigm, we will no doubt find the style presented here as archaic as we find today machine code of the 1950s.

15.2.3 Parsing with VW Grammars

Parsing with VW grammars is an interesting subject. On the down side it cannot be done: it can be proved that there cannot be a general parser/processor for VW grammars. On the up side, with some reasonable restrictions, a lot can be done.

There are several known techniques; we mention here the CF-skeleton technique, the definite clause/Prolog technique, and LL(1) parsing, but the literature references in (Web)Section 18.2.6 contain additional descriptions. The Definite Clause/Prolog technique is the most convenient, and we will discuss it here in some depth. We will also briefly introduce the CF-skeleton technique. The LL(1) parsing technique is very interesting, very complicated and quite powerful; we refer the reader to the papers by Gerevich [258] and Fisher [263, 273].

In the CF-skeleton technique a skeleton grammar is extracted from the VW grammar by ignoring the metanotions, so the hypernotions reduce to simple CF non-terminals. An essential ingredient for this is an algorithm for finding out whether a given hypernotion, occurring in the right-hand side of a hyperrule, can ever match a given hyperrule. For example, for the above grammar the algorithm should be able to find out that the **`write N1`** from the hyperrule **`produce N1 times i: write N1`** can be expanded into something that can be matched by the hyperrule **`write N i: write N, i symbol`**. One says that the algorithm should solve the *cross-reference problem*. This seems easy enough for our examples, but it can be proved that no such algorithm can exist: the cross-reference problem for VW grammars is unsolvable. But, as usual, with some ingenuity one can construct an approximation algorithm, or one can impose restrictions on the grammar.

When we apply the CF-skeleton transformation to the grammar from Figure 15.5 it vanishes almost completely (which immediately shows that such techniques do not work for every VW grammar), so we turn to the less "vehemently two-level" grammar from Figure 15.4 for an example. A likely result would be the skeleton grammar (in CF notation)

```
text_s:  A B C.
A: a.    A: a A.
B: b.    B: b B.
C: c.    C: c C.
```

The input is then parsed using this CF grammar and any suitable CF parsing method; a CF parse forest results in which segments of the input are identified as produced by the CF remainders of the hypernotions. Various techniques are used to extract information about the metanotions from this structure (see for example Dembiński and Małuszyński [254]). This information is then checked, used to reduce the number of trees in the parse forest, and correlated with the resulting tree(s) to yield the semantics.

The Prolog approach also uses a CF skeleton grammar and converts it to a Prolog program using the Definite Clause technique, as explained in Section 6.7. In the VW version of the Definite Clause technique, a Prolog rule is defined for each hyperrule. Its structure derives from the skeleton grammar and the names of the goals in it

correspond to the non-terminals in that grammar. The metanotions in the hyperrule are added as logic variables, in addition to the **S** and **R** resulting from the conversion to definite clauses.

This narrows down the "suitable CF parsing method" mentioned above to full backtracking top-down parsing, and the "checking and correlating" to unification of logic variables. Both features are built-in in Prolog, and are well studied and well understood.

For many VW grammars, the result is a reasonably understandable Prolog program which is a reasonably effective parser for the VW grammar. The translation of our grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ shown in Figure 15.8 is a good example. To avoid clutter

```
text(S,N,R):- a_n(S,N,R1), b_n(R1,N,R2), c_n(R2,N,R).

a_n(S,[i],R):- symbol(S,a,R).
a_n(S,[i|N],R):- symbol(S,a,R1), a_n(R1,N,R).

b_n(S,[i],R):- symbol(S,b,R).
b_n(S,[i|N],R):- symbol(S,b,R1), b_n(R1,N,R).

c_n(S,[i],R):- symbol(S,c,R).
c_n(S,[i|N],R):- symbol(S,c,R1), c_n(R1,N,R).

symbol([A|R],A,R).
```

Fig. 15.8. A recognizer for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ in Prolog

we have abbreviated **Sentence** to **S** and **Remainder** to **R**, and we have replaced clause names that start with a capital letter like **A** by a form like **a_n**, as we did in Section 6.7. Presented with the query

```
| ?- text([a,a,a,b,b,b,c,c,c], N, []).
```

the system answers

```
N = [i,i,i]
```

and to the query **text([a,a,a,b,b,b,c,c,c], N, [])** it answers **no**.

Surprisingly, even the VW grammar from Figure 15.5, which had a CF skeleton grammar with only one, nameless, non-terminal, leads to a Definite Clause program that works, as Figure 15.9 shows. We have named the corresponding Prolog rule **x**.

(There are the usual real-world problems with this approach; for example the rule name **c** in Figure 15.8 must be changed before running the program because **c** is a system predicate in Cprolog and cannot be redefined.)

Parsing time requirements are exponential in principle, or even infinite, but for many grammars the parser runs in linear time. As usual in all top-down parsers, left recursion causes problems; see the section on cancellation parsing (Section 6.8) for possible solutions.

```
text(S,N,R):- x(S,a,N,R1), x(R1,b,N,R2), x(R2,c,N,R).

x(S,A,[i],R):- symbol(S,A,R).
x(S,A,[i|N],R):- symbol(S,A,R1), x(R1,A,N,R).

symbol([A|R],A,R).
```

Fig. 15.9. An even shorter recognizer for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ in Prolog

Edupuganty and Bryant [262] implement a similar parser that is independent of Prolog or DCGs.

15.2.4 Error Handling in VW Grammars

Error handling is a problem. As with any backtracking system, when a Definite Clause parser cannot find what it is looking for, it backtracks, and when whatever it is looking for is not there, it backtracks all the way: it returns to the initial position and proudly announces "No!"; see Section 7.1 and especially Figure 7.5. This is not very helpful in debugging the grammar or in finding an error in the input. It is not even possible to add a logic variable to obtain the longest prefix of **Sentence** that the system has managed to match: the successive failures will uninstantiate this variable again and again, and it will not be set when the parser fails. In the end the trace facility of the Prolog system will have to come to the rescue, an often effective but always unelegant solution.

15.2.5 Infinite Symbol Sets

In a sense, VW grammars are even more powerful than PS grammars: since the name of a symbol can be generated by the grammar, they can easily handle infinite symbol sets. Of course this just shifts the problem: there must be a (finite) mapping from symbol names to symbols somewhere. The VW grammar of Figure 15.10 generates sentences consisting of arbitrary numbers of equal-length stretches of equal symbols, for example, $s_1s_1s_1s_2s_2s_2$ or $s_1s_1s_2s_2s_3s_3s_4s_4s_5s_5$, where s_n is the representation of the $\mathtt{i}^n$ **symbol**. See Grune [274] for more details.

```
N ::          n N; ε.
C ::          i; i C.

text_S:       N i tail.
N C tail:     ε; N C, N C i tail.
N n C :       C symbol, N C.
C :           ε.
```

Fig. 15.10. A grammar handling an infinite alphabet

15.3 Attribute and Affix Grammars

The desire to bridge the gap between VW grammars — very powerful but next to unparsable — and CF grammars — annoyingly lacking in power but quite parsable — gave rise to two developments, one down from VW grammars and one up from CF grammars. The first yielded affix grammars (Koster and Meertens [237]), the second attribute grammars (Knuth [243]). They meet in the middle in the EAGs, (Watt and Madsen [259]) which may stand for Extended Attribute Grammars or Extended Affix Grammars.

Although a comparison of attribute and affix grammars finds far more similarities than differences (Koster [269]), their provenance, history and realm of application differ so much that it is useful to treat them in separate sections.

15.3.1 Attribute Grammars

An *attribute grammar* is a context-free grammar extended with two features. First, the non-terminals, and usually also the terminals, in the grammar have values from some programming language P attached to them; these values are called "attributes". And second, each grammar rule has attached to it a piece of code in the language P, its *attribute evaluation rule*, for establishing these values.

15.3.1.1 Attributes and Evaluation Rules

All rules for a non-terminal N specify the same set of attributes; each grammar rule R for N specifies its own evaluation rule. We will see later on why this has to be so. The attributes and their evaluation rules have to be supplied by the grammar writer. There are two forms of evaluation rules: functions and checks. An evaluation rule for a rule R for a non-terminal N can express some attribute A of N or of one of its children in R as functions of some other attributes, thus expressing a part of the semantics of the rule R:

$$A := func_{A,R}(A_p, \ldots, A_q);$$

where $func_{A,R}$ is the function in rule R responsible for the evaluation of attribute A. The evaluation code can also check some attributes of N in order to impose context-sensitive restrictions:

$$check_R(A_p, \ldots, A_q);$$

In some systems, the evaluation rules can also steer the parsing; their use is then similar to that of the conflict resolvers from Section 8.2.5.3.

We see that again two levels are involved, a CF level to express the syntax and an attribute evaluation rule level to express the context dependencies; as in VW grammars, the second level can at the same time express the semantics. We will now look in more detail at the use of the attributes and their evaluation rules.

Although the grammar rules for a non-terminal N are written only once in the grammar, many nodes for N, possibly stemming from different rules for N, can be

present in the parse tree. Each of these nodes has room for the same set of attributes; their values can, and usually will, differ from node to node. These values can be anything the programming language allows: numbers, strings, etc. For a given node for *N*, the attribute evaluation rules compute the values of some attributes of *N* and of some attributes of *N*'s children, using other attribute values of *N* and its children. So the rules provide local communication between the attributes of *N* and those of its children; and since *N* has a parent unless it is the root, and *N*'s children have again children unless they are terminal symbols, this scheme allows the computation of relationships all throughout the parse tree, between the terminal symbols and the root. As we have seen, these relationships can be used to implement context-sensitive conditions, and to compose semantics.

If the attribute evaluation rules of a grammar rule *R* for *N* compute an attribute of the left-hand side of *R*, that attribute is a *synthesized attribute* of *N*; all evaluation rules for *N* must have functions for the computation of all synthesized attributes of *N*. (Synthesized attributes are also known as *derived attributes*.) If the attribute evaluation rules compute an attribute of one of the non-terminals *A* in the right-hand side of *R*, then that attribute is an *inherited attribute* of *A*, and all grammar rules that have *A* in their right-hand side must have functions for the computation of all inherited attributes of *A*. Terminal symbols have their own built-in synthesized attributes; for example, a token `3` could have a string-valued synthesized attribute with value `"3"` or an integer-valued synthesized attribute with value 3 or 51 (its ASCII code). This way there is a function for the computation of each attribute in each node in the parse tree. The synthesized attributes of the root can be viewed as the semantics of the entire input string.

Figure 15.11 shows an attribute grammar for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, where the attribute evaluation rules are given in curly brackets, and **`syn`** and **`inh`** indicate the modes of the attributes. The notation used is ad hoc, but sufficient for our purposes; existing sys-

```
text_s(syn int n):  A(na) B(nb) C(nc)  {nb:=na; nc:=na; n:=na}.

A(syn int n):       'a'                {n:=1;}.
A(syn int n):       'a' A(na)          {n:=na+1;}.

B(inh int n):       'b'                {check(n==1);}.
B(inh int n):       'b' B(nb)          {check(n>1); nb:=n-1;}.

C(inh int n):       'c'                {check(n==1);}.
C(inh int n):       'c' C(nc)          {check(n>1); nc:=n-1;}.
```

Fig. 15.11. Attribute grammar for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

tems have much more elaborate notations. Rather than letting the rules for **`A`**, **`B`**, and **`C`** synthesize the number of letters they collect and then check the equality of these numbers at the top level, we let **`A`** do the synthesizing, pass the resulting value to **`B`**

and **C** as inherited attributes, and let these two check when they recognize their last token. This has the advantage of giving an error message on the first possible token.

15.3.1.2 Parsing and Attribute Evaluation

Parsing can be done using any suitable CF method; for the above attribute grammar LL(1) could be used. There are three ways to evaluate the attributes: bottom-up (data-driven), top-down (demand-driven), and very clever ("ordered"). We will briefly discuss the first two methods, using the input string **aabbcc**. The CF parse tree is given in Figure 15.12.

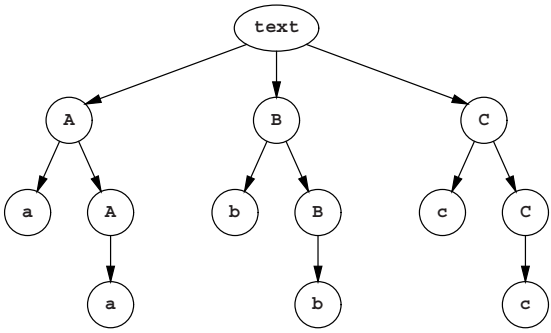


Fig. 15.12. The not yet attributed CF parse tree for **aabbcc**

In the bottom-up method, only the attributes for the leaves are known initially, but as soon as sufficient attributes in a right-hand side of a grammar rule are known, we can use its attribute evaluation rule to compute an attribute of its left-hand side or of a child. Initially only the attribute **n** of the node **A(n):'a'** is known. This allows us to compute the **n** in **A(n):'a'A(na)**, which in turn gives us the attributes of **n**, **na** and **nb** of **text(n):A(na)B(nb)C(nc)**. Since the **nb** is the inherited attribute **n** of **B(n):'b'B(nb)**, the next bottom-up scan will be able to compute the attribute **nb** of that rule. This way the attribute values (semantics) spread over the tree, finally reach the start symbol and provide us with the semantics of the whole sentence. If there are inherited attributes, repeated scans will be needed, so this method is primarily indicated for attribute grammars with synthesized attributes only.

In the top-down method, we demand to know the value of the attribute **n** of the root **text**. Since this is computed by **n:=na** and we do not know **na** yet, we have to descend into the tree for **A**, meeting assignments of the form **n:=na+1** and postponing them, until we finally reach the assignment **n:=1**. We can then do all the postponed assignments, and finally come up with the value of **n** in **text**. So, as far as semantics is concerned, we are finished now, but if we want to do checking, we have to evaluate (top-down) the arguments of the **check**s. (The large discrepancy between checking and semantics here is an artifact of the highly redundant input.)

These two methods are characterized by a fixed and grammar-independent evaluation order of the attribute code. For ordered attribute grammars (Kastens [255]) an optimal evaluation order can be derived from the grammar, which allows very efficient attribute evaluation. Constructing such evaluation orders is outside the scope of this book; see, for example, Grune et al. [414, Ch. 3].

The fully attributed tree is shown in Figure 15.13. The arrows show the directions in which the information has flowed; the checks are not shown.

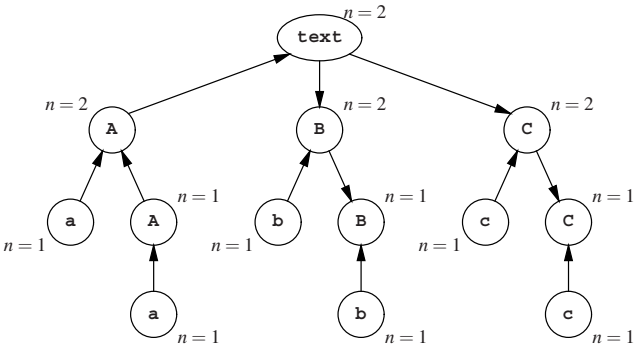


Fig. 15.13. The fully attributed parse tree for input **aabbcc**

The error handling in attribute grammars can be divided in two parts, the syntactic error handling, which comes with the CF parsing method, and the context-sensitive error handling. The latter is implemented in the checks in the attribute evaluation rules attached to the rules, and is therefore fully the responsibility of the programmer. This division of responsibilities, between automation and user intervention, is characteristic of attribute grammars.

We have seen that attribute grammars are a successful mix of CF parsing, a programming language, context-dependencies, establishing semantics, and conflict resolvers. What *can* be automated *is* automated (CF parsing and the order in which the attribute evaluation rules are performed), and for the rest the grammar writer has almost the full power of a programming language.

Attribute grammars constitute a very powerful method of handling the CF aspect, the context-sensitive aspect and the semantics of a language, and are at present probably the most convenient means to do so. It is doubtful, however, that they are generative grammars, since it is next to impossible to use them as a sentence *production* mechanism; they should rather be classified as recognition systems.

15.3.2 Affix Grammars

Like attribute grammars, *affix grammars* (Koster [245, 269]) endow the nonterminals with parameters, called *affixes* here; and like attributes these are divided into synthesized, called "derived" here, and inherited. But where attribute grammars

have evaluation rules which express checks and semantics and which have no direct grammatical significance, affix grammars have *predicates*, which are a special kind of non-terminals which can only produce {ε}, the set containing only the empty string, or the empty set {}, which contains nothing. This allows affix grammars to be used as generative grammars.

15.3.2.1 Producing with an Affix Grammar

During the production process, a non-terminal can be replaced by any of the strings that are in the set of its terminal productions, and the same applies to predicates. If a predicate *P* has {ε} as the set of its terminal productions, that set contains only one item, ε, so *P* gets replaced by it, disappears, and the production process continues. But if *P* has {} as the set of its terminal productions, *P* cannot be replaced by anything, and the production process halts. Note that this hinges on the difference between the empty set and a set containing one element, the empty string.

Figure 15.14 shows how this mechanism controls the production process. The

```
start(δ int n):  A(n) B(n) C(n).

A(δ int n):      n = 1, 'a'.
A(δ int n):      'a', A(na), n = na+1.

B(ι int n):      n = 1, 'b'.
B(ι int n):      n > 1, 'b', nb = n-1, B(nb).

C(ι int n):      n = 1, 'c'.
C(ι int n):      n > 1, 'c', nc = n-1, C(nc).
```

Fig. 15.14. Affix grammar for the language $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$

forms **n=1**, **nb=n-1**, etc. are predicates; they produce ε when the affixes obey the test, and the empty set otherwise. In this figure δ indicates a derived affix and ι an inherited one, but the difference plays no role during production. To produce **aaabbbccc** we begin with **start(3)**. This leads to **A(3)**, but the first rule for **A** fails since the predicate **n=1** produces nothing. The second rule produces **a** followed by **na** **a**s, but production can only continue if we choose **na** to be 2, to pass the predicate **n=na+1**. Related considerations apply to the production of the segments **bbb** and **ccc**.

The predicates **n>1** are not really necessary: entering the rule for **B** with **n** equal to 1 would cause **nb** to be forced to 0, and attempts to produce from **B(0)** would lead to an infinite dead end. We have blocked these infinite dead ends for esthetic reasons.

15.3.2.2 Parsing with an Affix Grammar

Although the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ is symmetrical in **a**, **b**, and **c**, the grammar in Figure 15.14 is not. The reason is that it is a *well-formed affix grammar*, which means that the form of the predicates, their positions, and the δ and ι information together allow the affixes to be evaluated during parsing. The precise requirements for well-formedness are given by Koster [245], but effectively they are the same as for attribute evaluation in attribute grammars.

All the usual general parsing techniques are possible, and they handle a failing predicate just as bumping into an unexpected token. A simple top-down parser with **aaabbbccc** as input would first try the first rule for **A**, derive **n** equal to 1 from it, continue with **B(1)**, find there is no **b**, backtrack, try the second rule for **A**, try **na** equal to 1, fail, backtrack, etc., until all 3 **a** are consumed. The derived value of **n** of the top-level **A** is then 3, which is passed as inherited affix to **B** and **C**. The rest is straightforward.

Watt and Madsen [259] have extended the affix grammars with a transduction mechanism, resulting in the EAGs. These can be viewed as attribute grammars with generative power, or as affix grammars with semantics.

15.3.2.3 Affix Grammars over a Finite Lattice

Affixes can be of any type, but an especially useful kind of affix grammar, the *AGFL*, is created by restricting the affixes to finite lattices (Koster [268]). A lattice is a set of values that can be compared for rank; we shall use $\succ$ as the ranking operator. If x and y are compared for rank, the answer may be "smaller", "larger", or "not ordered"; it may also be "equal" but then x and y are the same value. An important condition is that there cannot be a value x for which we have $x \succ \cdots \succ x$. A lattice corresponds to a directed acyclic graph, a "dag". A formal definition of AGFLs is given by Nederhof and Sarbo [349].

Such lattices are very useful for encoding attributes of linguistic forms, like gender, tense, active/passive, etc. A simple example is gender in, for example, German, Russian, and several other languages:

```
GENDER :: masculine; feminine; neuter.
```

Here **GENDER** $\succ$ **masculine**, **GENDER** $\succ$ **feminine**, and **GENDER** $\succ$ **neuter**. But the usefulness of lattices is demonstrated better in a language that has a more complicated gender structure. One such language is Burushaski, which has four genders (Figure 15.15), **hm** (human masculine), **hf** (human feminine), **x** (countable non-human) and **y** (non-countable); examples of non-countable are "salt", "love", etc. This structure is shown in the first group of affix rules for **GENDER**; **hm**, **hf**, **x**, and **y** are affix terminals; the upper case words are affix non-terminals; and the **::** identifies the rules as affix rules.

But some parts of the Burushaski grammar treat feminine nouns quite differently from the rest, so a division in feminine and non-feminine is also appropriate; see the second group of affix rules. The dag for this affix type is shown in Figure 15.16

```
GENDER ::        COUNTABLE; UNCOUNTABLE.
COUNTABLE ::     HUMAN ; x.
HUMAN ::         hm ; hf.
UNCOUNTABLE ::   y.

GENDER ::        FEMININE; NONFEMININE.
FEMININE ::      hf.
NONFEMININE ::   hm; x; y.
```

Fig. 15.15. Finite Lattice affix rules for gender in the Burushaski language

in traditional lattice representation: if v_1 is higher than v_2 and connected by lines, then $v_1 \succ v_2$; the mathematical definition of lattices requires a top element ($\top$) and a bottom element ($\bot$). It will be clear why lattices are called lattices.

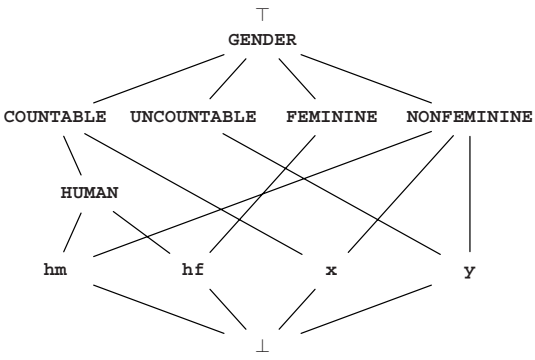


Fig. 15.16. Lattice of the FL affix notion **GENDER** of Figure 15.15

The following (very naive) grammar fragment shows the use of the affixes:

```
noun phrase (GENDER) :
      [article (GENDER)], noun (GENDER), infix (GENDER).
...
noun (HUMAN) : family member (HUMAN).
...
infix (hf) : "mu" .
infix (NONFEMININE) : .
...
```

The first rule says that gender in a noun phrase distributes over an optional article, the noun, and the infix. As in VW grammars, the affix variable must be substituted consistently. If the actual gender is in the subset **HUMAN**, the noun may be a family member of the same gender. If the gender is **hf**, the infix must be **mu**, but if it is in the subset **NONFEMININE** it must be left out.

Since a lattice does not contain cycles, a grammar which is a lattice can produce only a finite number of terminal productions: it is a Type 4 grammar (page 33). This

means that one could substitute out all affixes in an AGFL and obtain a context-free grammar, but such a grammar would be much larger and less convenient. So an AGFL is much weaker than an general affix grammar, but its strength lies in its ease of use for linguistic purposes and in its compactness.

AGFLs can be parsed by any CF method. The problem lies in managing the affixes: naive methods cause an explosive growth of the data structures. An efficient solution is given by Nederhof and Sarbo [349].

15.4 Tree-Adjoining Grammars

As languages go, English is a rather context-free language in that it exhibits few long-range relationships. Verbs do correlate with the subject ("I walk" versus "he walks"), but the subject is seldom far away. It has composite verbs (for example "to throw away"), but the two parts usually stay close together: "She finally threw the old towels, which she had inherited from her grandmother and which over the years had assumed paler and paler shades of gray, away" is awkward English. "The brush I painted the garage door green with .." (where "with" relates back to "brush") is better, but many speakers would prefer "The brush with which . . . ".

This is not true for many other languages. Verbs may agree with subjects, direct and indirect objects simultaneously in complicated ways (as in Georgian and many American Indian languages), and even languages closely related to English have composite verbs and other composite word classes, the parts of which can and often must move arbitrarily far away from each other. Examples are Dutch and Swiss German, and to a lesser degree Standard German. Especially the first two exhibit so-called *cross-dependencies*, situations in which the lines that connect parts of the same unit cross. Such relationships do not correspond to a tree, and CF grammars cannot produce them. Cross-dependencies have long worried linguists, and if linguistics want to explain such languages in a generative way, an alternative system is needed. *Tree Adjoining Grammars* (or *TAGs*) is one such system; it was developed by Joshi [250]. Yngve [375] describes a simpler and much earlier system.

15.4.1 Cross-Dependencies

As an example of a cross-dependency we will take the Dutch sentence

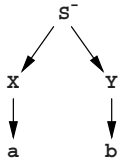
```
Daar doe ik niet aan mee.
```

This is a completely normal everyday Dutch sentence, but its structure is complex. It contains the words **ik** = "I", **niet** = "not", **meedoen** = "participate", and **daaraan** = "to that", and it means "I do not participate in that", or more idiomatically "I will have no part in that". (Since English has no cross-dependencies, Dutch sentences that use it have no literal translation in English.) We see that the words **daaraan** and **meedoen** have split up, and the dependencies connecting the parts cross:

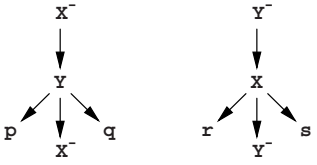
It is clear that there cannot be a CF grammar which produces this sentence so that **daar** and **aan** derive from one non-terminal and **doe** and **mee** derive from another. (We will not go into the fact that there is more than one kind of cross-dependency, nor into their linguistic relevancy here.)

TAGs were designed to solve this problem. We shall first explain the principles and then see how to create a TAG that will produce the above sentence in a satisfactory way. Where a CF grammar has rules, a TAG has trees. A CF rule names a non-terminal and expresses it in terminals and other non-terminals, and a TAG tree names a node and expresses it in terminals and other nodes. The top node of a tree is labeled with the name of the tree, and all the internal nodes are labeled with non-terminals; the leaves are usually terminals but can occasionally be labeled with a non-terminal. If a tree has terminal leaves only, it is a *terminal tree*, and it represents a string in the language generated by the TAG; that string is the sequence of terminals spelled by the leaves. An example of such a terminal tree will be shown in Figure 15.19.

Whereas a CF grammar has one start symbol, a TAG has several start trees called *elementary trees*. They usually represent the basic sentence types of a language: active sentences, passive sentences, questions, etc., which is why they are also called *sentential trees*. All the leaves of an elementary tree are terminals, so an elementary tree might look like this:

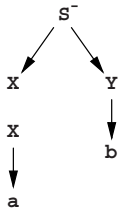


and it represents the string **ab**. (The meaning of the **-** marker next to the **S** will be explained below.) The other trees in the grammar (called *auxiliary trees*) have the same structure as elementary trees, except that one leaf is not a terminal but is labeled with the name of the tree:

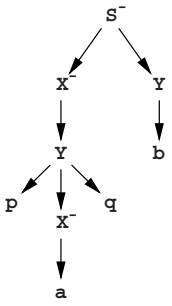


This leaf is called the "foot" of the tree. Note that there is exactly one path from the top of the tree to its foot.

Auxiliary trees do not represent strings in the language, but serve to expand nodes in elementary trees in a process called *adjoining*, as follows. Suppose we want to expand the node **X** in the elementary tree **S** by using the tree for **X**. We first cut the node **X** in **S** in two to detach its subtree from **S**:



Next we put the tree for **X** in between and connect the top to the open **X** node in **S** and the foot to the detached subtree:



Since tops and feet are labeled identically, this does not disrupt the structure of the tree. The result represents the string **paqb**. Since the **.a.b** come from one tree and the **p.q.** come from another, we have already created our first cross-dependency! And we can also see how it works: the tree **X** "spreads its wings" to the left and the right of the **a**, thus creating a (moderately) long-range correlation between its left wing and its right wing.

Some nodes in the above pictures are marked with a **-** marker; this marker indicates that the marked node cannot be expanded. We shall also meet nodes marked with a **+**; such nodes *must* be expanded for a tree to count as a terminal tree. Other varieties of TAGs may define other types of markers with their requirements.

We now turn to the sample sentence "**daar doe ik niet aan mee**", and start by creating a tree for **daar ... aan**. Sentences that do not start with the subject have the form of an "inverted sentence", one in which the verb precedes the subject; we will make a tree called **INV_S** for these. So the sentence consists of four pieces:

S = daar INV_S$_l$ aan INV_S$_r$

where **INV_S**$_{\mathrm{l}}$ is the left wing of **INV_S** and **INV_S**$_{\mathrm{r}}$ is its right wing. What is missing to turn this into an auxiliary tree is the position of the foot **S**. To determine that position, we need information about the language. In Dutch, no words can be inserted between **aan** and **INV_S**$_{\mathrm{r}}$, but **INV_S**$_{\mathrm{l}}$ and **aan** can be separated by words. So that gives us the proper place for the foot:

S = daar INV_S$_{\mathrm{l}}$ **S**$_{\mathrm{foot}}$ **aan INV_S**$_{\mathrm{r}}$

This leads to the middle tree in Figure 15.17, where the **-** markers show that after adjoining the two nodes **S** cannot be adjoined again, and the **+** shows that **INV_S** must be adjoined to obtain a sentence. Figure 15.17 also shows the elementary tree **S** to start the process and the result of adjoining the tree for **S** to it.

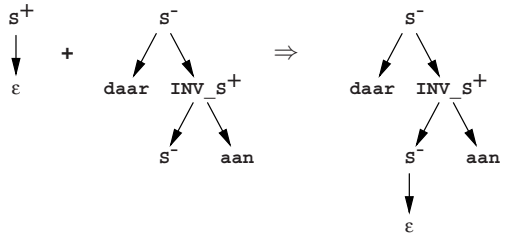


Fig. 15.17. Elementary tree, tree for **S** and result of adjoining

In a real grammar there will be many trees for **INV_S**; we construct one here for a negative inverted sentence. It contains an inverted verb part **INV_VP**. So the remainder of the sentence

INV_S = doe ik niet INV_S$_{\mathrm{foot}}$ **mee**

corresponds to

INV_S = INV_VP$_{\mathrm{l}}$ **niet INV_S**$_{\mathrm{foot}}$ **INV_VP**$_{\mathrm{r}}$

The tree is shown in Figure 15.18(*a*). This leaves

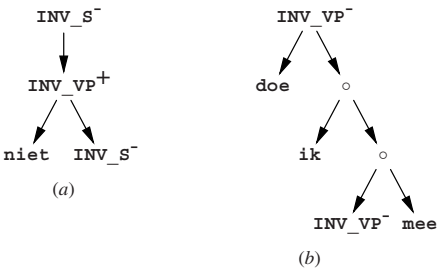


Fig. 15.18. Trees for **INV_S** and **INV_VP**

doe ik INV_VP$_{\text{foot}}$ **mee**

for **INV_VP**. The three words in it are strongly interdependent, so for simplicity we will accept this segment as a single tree. The tree is shown in Figure 15.18(*b*) and shows two anonymous nodes; if a particular formalism does not allow this, dummy names could be assigned to them. Figure 15.19 shows the complete tree; we see that it contains no nodes marked with a **+**, and that its outer rim spells

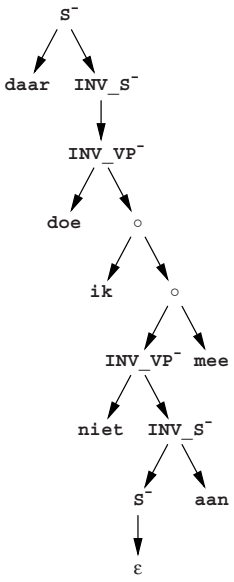


Fig. 15.19. The complete tree for the Dutch sentence with cross-dependency

"**daar doe ik niet aan mee**".

In a real-world system the tree for **doe ik ... mee** would be split up into one for a subject **ik** and one for a conjugated verb **doe ... mee**, but since the verb agrees with the subject (**doe ik ... mee** versus **doet hij ... mee**, etc.) another type of correlation must be established. In many systems, the trees and the nodes in them are provided with attributes, often called *features* in linguistics, that are required to agree for adjoining to be allowed. Since these attributes have only a finite (and usually very small) number of values, all the trees with all their values could in principle be written out, and the attributes are a means to complexity reduction only, just as a finite-choice metagrammar is to a VW grammar.

The TAG design process shows clearly how each detail of Dutch syntax is expressed in a particular auxiliary tree. There is a large TAG for the English language available on the Internet from the University of Pennsylvania.

TAGs are (somewhat) stronger than CF grammars, since they can produce the language $a^n b^n c^n$, which is not CF, but they are (much) weaker than CS grammars.

They cannot, for example, produce the language $a^n b^n c^n d^n e^n$. Remarkably, TAGs can create only 4 copies of the same number.

The semantics of a Tree Adjoining Grammar is attached to the trees, and composition of the semantics is straightforward from the adjoining process.

15.4.2 Parsing with TAGs

Input described by a TAG can be parsed in polynomial time, using a bottom-up algorithm described by Vijay-Shankar and Joshi [264]. The algorithm, which is very similar to CYK (Section 4.2), is not difficult, but since TAGs are more complicated than CF grammars there are many more details, and we will just sketch the algorithm here. We assume that no node in any elementary or auxiliary tree has more than two children; it is easy to get a TAG into this "2-form" by splitting nodes where necessary.

Rather than having a two-dimensional recognition table $R_{i,l}$ the entries of which contain non-terminals that produce $t_{i..i+l-1}$ where $t_{1..n}$ is the input string, we have a four-dimensional recognition table $A_{i,j,k,l}$ the entries of which contain tree nodes X in trees for Y that produce the segments $t_{i..j}$ and $t_{k..l}$, where the gap $t_{j+1,k-1}$ between them is to be produced by the tree hanging from the foot node of Y. Note that in the description of CYK we used starting position and length to describe a segment; here it is more convenient to use the start and end positions of both segments.

The meaning of the entry $A_{i,j,k,l}$ is shown in Figure 15.20; the drawn lines de-

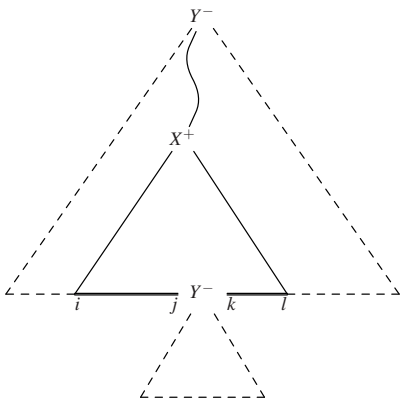


Fig. 15.20. Recognized footed tree for the node X in a tree for Y

marcate the recognized region, the broken lines show the region that must still be recognized to fully recognize a node of type Y. Note that it is the lower part of rule Y that has been recognized, in accordance with the bottom-up nature of the CYK algorithm. It is a fully recognized subtree, except that one of its leaves is the foot of Y; we shall call such subtrees "footed trees". The whole input string $t_{1..n}$ is recognized when at least one of the entries $A_{1,j-1,j,n}$ contains the root of an elementary tree, for

any value of j. Once the table has been filled in, we can find parse trees by working from the contents of the $A_{1,j-1,j,n}$ downwards, in a way similar to that of Section 4.2.5.

The actual contents of the entries in the table A are paths of the form $Y \cdots X$, where Y is the top of a tree from the grammar, and X is the node carrying the footed tree. Such a path can, for example, be implemented by giving the start node Y and a string of left-right-left directives that tell how to reach the right X from Y. Including Y is necessary to allow us to adjoin a completed tree to the foot of the part recognized by X, and including the explicit path is necessary because Y may contain more than one occurrence of X.

The essential step in the CYK algorithm is the combination of two recognized regions into a third, larger region. Doing these steps in the right order allows one to fill the table A in one sweep, visiting and filling the entries by combining the contents of entries that have already been filled. The table has n^4 entries, so filing it costs n^4 times the cost of the actions for one entry.

In TAGs there are two fundamental combination steps, one side-ways and one upwards. There are several other combination steps, but they are the mirror images of the fundamental ones, are special cases for elementary trees, or are trivial.

The most characteristic combination step for TAGs is the upwards combination, shown in Figure 15.21. It extends a footed tree for the path $Y \cdots X$ with a completely

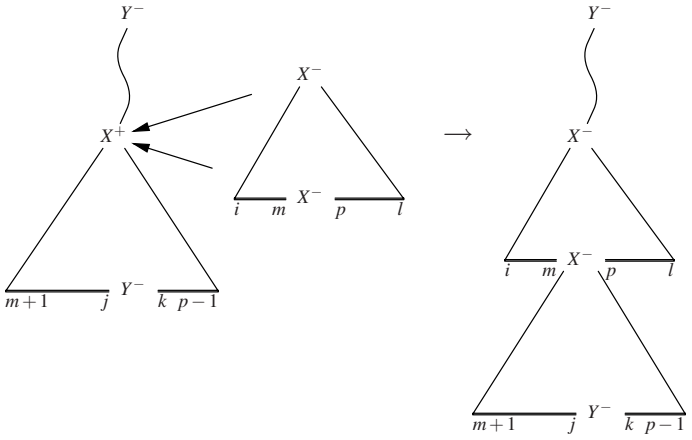


Fig. 15.21. The upwards-combination step for Tree Adjoining Grammars

recognized auxiliary tree for X, to form a new, larger, footed tree for the path $Y \cdots X$. The completely recognized tree for X covers the sections $t_{i..m}$ and $t_{p..l}$; we can find out that a tree is completely recognized from its entry in the table A, which has the form $X \cdots X$, that is, the length of the path is 0. The footed tree for $Y \cdots X$ must then start at position $m+1$ and end at $p-1$. Suppose its foot spans a gap from j to k; then the combined footed tree, still identified by $Y \cdots X$, can be inserted in $A_{i,j,k,l}$. So, to fill the entry $A_{i,j,k,l}$ with all results of the upwards combination step, we must search

for values for m and p such that the conditions of Figure 15.21 are fulfilled. There can be at most $O(n)$ such values for each of the two variables, so this step costs at most $O(n^2)$ actions per entry.

The sideways combination increases the size of a not completely recognized tree, as shown in Figure 15.22. It concentrates on those nodes Z in trees Y that have one already recognized child which a Y-footed tree $Y \cdots X$, and a second recognized child $Y \cdots W$, whose leftmost recognized token is next to the rightmost token of $Y \cdots X$; this happens at the $p, p+1$ junction in Figure 15.22. These two children can then combine

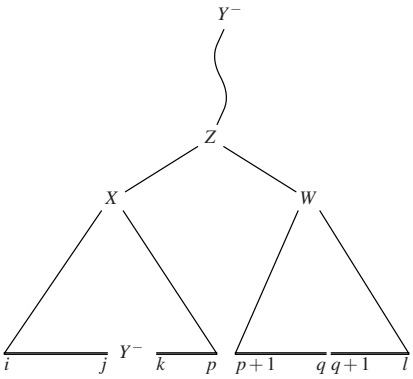


Fig. 15.22. The sideways-combination step for Tree Adjoining Grammars

sideways into a footed tree for $Y \cdots Z$. Of course $Y \cdots W$ cannot be a footed tree, since the auxiliary tree Y has only one foot. So, to fill the entry $A_{i,j,k,l}$ with all results of the sideways combination, we must search for values for p and q such that the conditions of Figure 15.22 are fulfilled. In principle there can again be at most $O(n)$ such values, so this step could cost $O(n^2)$ per entry. The value of q, however, is fairly arbitrary and can be taken out of the game by keeping a separate two-dimensional table $B_{i,l}$, containing copies of the nodes in $A_{i,p,p+1,l}$ for all values of p. All suitable nodes $Y \cdots W$ can now be found in $O(n)$ actions. The table also helps in finding top-level recognitions of the form E in $A_{1,p,p+1,n}$, where E is the root of an elementary tree. Although this will speed up our algorithm, in does not help for the overall complexity since the upwards combination step already takes $O(n^2)$.

We still have to answer the question where the initial values in the table A come from. They come from two sources, terminal symbols and empty footed trees. Each terminal symbol t directly attached to a node X in a tree for Y gives rise to an entry $Y \cdots X$ in $A_{i,i,i+1,i}$ for each position i in the input where t is found; that is, the token is absorbed in the left wing of the footed tree and the right wing is empty. And empty footed trees $Y \cdots Y$ for all auxiliary rules Y are entered in all $A_{i,i-1,j,j-1}$ for all $1 \leq i \leq j \leq n$; that is, footed trees for foot nodes with empty wings are recognized everywhere. At first they will grow sideways, using the finished subtrees originating from the terminal symbols.

We have now seen a general CYK-like parsing algorithm for TAGs in "2-form"; its time complexity is $O(n^6)$, because it has to fill in n^4 entries at a cost of at most $O(n^2)$ each. Satta [276] proves that if we can do general TAG parsing in $O(n^p)$, we can do Boolean matrix multiplication in $O(n^{2+p/6})$; note that for $p = 6$ this amounts to the standard complexities for both processes. Since Boolean matrix multiplication in time less than $O(n^3)$ is very difficult, it is probable that general tree parsing in time less than $O(n^6)$ is also very difficult.

Very little is known about error recovery for TAGs. Perhaps the techniques available for the CYK algorithm could be adapted.

CYK parsing is not the only possibility for TAGs. Schabes and Joshi [271] describe an Earley parser for TAGs, and Nederhof [281] and Prolo [282] explore LR parsers for TAGs, with their problems.

15.5 Coupled Grammars

Coupled grammars establish long-range relations by creating all parties to the relation simultaneously and keeping track of them as they go their separate ways. The non-terminals in a coupled grammar consist of fragments called *components*. During the production process, all the components of a non-terminal N must be replaced at the same time, using the same alternative for N. Suppose we have the sentential form

```
... N1 ... N2 ... N3 ...
```

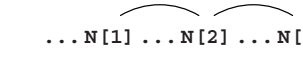
where $\mathbf{N}_1$, $\mathbf{N}_2$, and $\mathbf{N}_3$ are components that were created simultaneously, and the following grammar rule for N:

```
N1, N2, N3  ->  a P1 b, c d, e P2 f | Q1 R1 R2, R3, Q2 R4 Q3
```

where **P** consists of 2, **Q** of 3, and **R** of 4 components. Then two new sentential forms result from the simultaneous substitution process:

```
... a P1 b ... c d ... e P2 f ...
... Q1 R1 R2 ... R3 ... Q2 R4 Q3 ...
```

Th requirement that "$\mathbf{N}_1$, $\mathbf{N}_2$, and $\mathbf{N}_3$ were created simultaneously" is essential, since other occurrences of $\mathbf{N}_1$, $\mathbf{N}_2$, and $\mathbf{N}_3$ may be present in the sentential form, unrelated to the ones shown, and of course they are completely free in *their* substitution (as long as they obey their own consistent substitution) restriction. A better representation of the original sentential form would therefore be



and this is indeed how such a form must be implemented in a program.

Coupled grammars can easily express non-CF languages like $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$:

```
      Ss  ->  A1 A2 A3
A1, A2, A3  ->  a A1, b A2, c A3 | ε, ε, ε
```

and a coupled grammar for the language *ww*, where *w* is an arbitrary string of **a**s and **b**s, is trivial. They can also provide finite-choice abbreviations in other grammars, for example:

```
COND1, COND2, COND3, COND4  ->  if, then, else, fi
                            |   ( , | , | , )
                            |   si, alors, sinon, fsi
```

which allows the rule

```
conditional statement  ->  COND1 Boolean expression
                           COND2 statement sequence
                           COND3 statement sequence COND4
```

to produce various forms of the conditional statement in a programming language (but only the consistent ones!).

The power of coupled grammars depends on the number of components that one allows. It can be proved that for large numbers of components the power of coupled grammars approaches that of the CS grammars. There is little reason to put a hard limit on the number of components, so the attainable power is just under that of CS grammars.

15.5.1 Parsing with Coupled Grammars

Full backtracking top-down parsing works almost without modification for non-left-recursive coupled grammars. A sample parsing for the string **aabbcc** using the above grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ is shown in Figure 15.23 in a format similar to that of Figure 6.11. Note that the subscripts in the recognized (left) part of each snap-

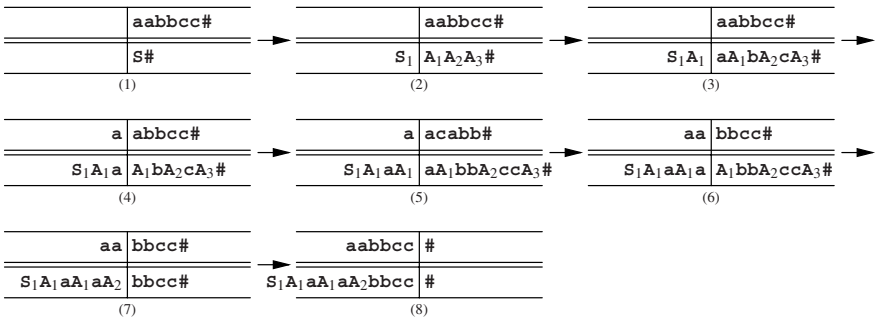


Fig. 15.23. Coupled-grammar parsing for the string **aabbcc**

shot represent the numbers of the chosen alternatives as they did in Figure 6.11, and those in the unrecognized (right) part represent numbers of components, as they do in a coupled grammar. So $\mathbf{A}_2$ on the left in the last snapshot identifies the rule $\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3 \rightarrow \varepsilon, \varepsilon, \varepsilon$ and $\mathbf{A}_2$ on the right in other snapshots identifies the second component of **A**. In the present case the backtracking is not activated (except for finding

out that there is no alternative parsing), but see Problem 15.10 for a parsing problem that requires backtracking.

The resulting parse tree, shown in Figure 15.24, is exactly the way one wants it, and semantics is attached easily.

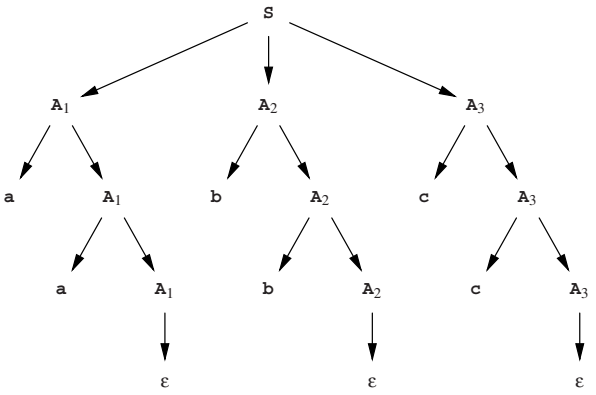


Fig. 15.24. The parse tree for the parsing of Figure 15.23

Pitsch [275, 277] gives algorithms for LL(1) and LR(1) parsing of coupled grammars, and Seki presents a general bottom-up parser based on CYK.

Coupled grammars were invented by Hotz in 1967 (Hotz [240, 278]) but received little publicity until the mid 1990s. They seem to be a convenient means of expressing mild context sensitivity, their main problem today being the lack of experience in their use.

15.6 Ordered Grammars

Ordered grammars produce non-CF languages by restricting the CF production process rather than by establishing long-range relationships. It is not clear whether this is a good idea in practice, but the ideas involved are certainly interesting, which is why we will discuss briefly two types of ordered grammars here.

15.6.1 Rule Ordering by Control Grammar

When using a CF grammar to produce a sentence, one is free to apply any production rule at any time, provided the required non-terminal is present in the sentential form. One type of ordered grammars restrict this freedom by requiring that the sequence of applied rules must obey a second grammar, the *control grammar*. Here, the rules in the original CF grammar are considered tokens in the control grammar, and a sequence of rules produced by the control grammar is called a *control sequence*.

It suffices to give just the control grammar, since the rules of the CF grammar are included in it.

Figure 15.25 gives an ordered grammar for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ in an EBNF notation. It produces, among many others, the control sequence

```
control_S  =>  [text_S -> A B C]
               ([A -> aA] [B -> bB] [C -> cC])*
               [A -> ε] [B -> ε] [C -> ε]
```

Fig. 15.25. An ordered grammar for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

```
[text_S -> A B C]
[A -> aA] [B -> bB] [C -> cC]
[A -> aA] [B -> bB] [C -> cC]
[A -> ε] [B -> ε] [C -> ε]
```

which in turn produces the final string **aabbcc**. Extension of the grammar to more than 3 tokens is trivial, and many other non-CF languages are also easily expressed. Usually a regular grammar is sufficient for the control grammar, as it was above.

The production process with ordered grammars can get stuck, even in more than one way. A control sequence could specify a CF rule for a non-terminal *A* that is not present in the CF sentential form. And when the control sequence is exhausted, the sentential form may still contain non-terminals. In both cases the attempted production was a blind alley.

15.6.2 Parsing with Rule-Ordered Grammars

Full backtracking top-down parsing, similar to the one explained for coupled grammars in Section 15.5.1, is sometimes possible for ordered grammars too. The basic action consists of making a prediction choice in the control grammar, which results in a rule to apply to the prediction for the rest of the input. If the rule cannot be applied or when the result contradicts the input, we backtrack over this choice and take the next one. If we can continue, we may get stuck further on, in which case we backtrack; or we may find a parsing based on this choice, and then if we need only one parsing we are done, but if we want all parsings we again backtrack.

The problem with this technique is that, just as in Definite Clause parsing, the process may not terminate. One reason is left recursion, but it can also happen that none of the prediction choices in the control grammar ever expands the leftmost non-terminal in the prediction, so the choices are never contradicted by the input, but the process does not get stuck either.

Since the input is recognized by the CF grammar, a normal parse tree results, to which semantics can be attached as to any CF parse tree.

For more information about this type of ordered grammars, see Friš [242] and Lepistö [247].

15.6.3 Marked Ordered Grammars

Although the "token" in the control sequence dictates what kind of non-terminal, for example **A**, is going to be substituted next during the production process, the system does not specify exactly which **A** is meant, in case there is more than one. This gives a certain non-determinism to the production process that seems to be alien to it: the control sequence does not really control it all. This is remedied in a variant described by Kulkarni [279], as follows. In a *marked ordered grammar* one member in the right-hand side of each CF rule is marked as the next one to be substituted. The control sequence must then provide the proper rule, and must end exactly when we find a marked terminal or a marked ε; otherwise the production is a blind alley.

Now the control grammar is in full control, but it is fairly clear that this cannot be the whole story: what about the other, non-marked non-terminals that may appear in right-hand sides and therefore in sentential forms? The answer is that a new control sequence is started for each of them.

Figure 15.26 gives a marked ordered grammar for the language $\mathbf{a}^n\mathbf{b}^{3n}\mathbf{c}^n$, where the marked members are between square brackets; rather than including the rules

```
CF grammar:
0:        S_S  ->  a[S]
1:         S   ->  [T]
2:         T   ->  B[T]c
3:         T   ->  [ε]
4:         B   ->  b [B]
5:         B   ->  [b]

control grammar
  control_S  =>  A 3 | 4 4 5
         A   =>  1 | 0 A 2
```

Fig. 15.26. A marked ordered grammar for the language $\mathbf{a}^n\mathbf{b}^{3n}\mathbf{c}^n$

in the control grammar, we have named them **0** to **5** and refer to them by those names in the control grammar. For demonstration purposes the stretches of 3 **b**s are produced by a second control sequence. The start symbol $\mathbf{control}_S$ generates two sets of control sequences, $\mathbf{0}^n\mathbf{12}^n\mathbf{3}$ and **445**. The sentential form starts as **S**, which is unmarked and so starts a new control sequence; **445** works on **B** only and would lead to a blind alley, so let us choose **001223**. This produces

aaBBεcc

We see that the nesting in the control grammar rule **A=>0A2** forces equal numbers of **a**s, **B**s and **c**s to be produced. Next, the **B**s are developed, using new control sequences. The control sequence **445** applies twice, and the end result is **aabbbbbbcc**, as expected.

There is a different way in which the CS restriction on the language can be viewed: the CF grammar produces strings with parse trees, but a parse tree is only acceptable when the path following marked nodes downwards from each non-marked node spells a word in the control language. This view is depicted in Figure 15.27, where the thicker lines represent the paths that have to obey the control grammar. We see that the long middle path correlates the left part of the tree with the right

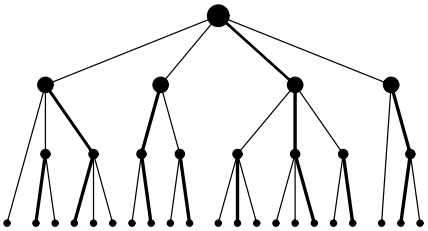


Fig. 15.27. Marked ordered grammar restrictions on a parse tree

part. The shorter paths arrange correlations in the subtrees only, and the subtrees of these are controlled by still shorter paths, and so on, so that control is distributed in an almost fractal way. The main path can, for example, correlate the subject, verb and object in a sentences; shorter paths can then correlate the adjectives in the subject with the noun in the subject, and the adjectives in the object with the noun in the object, as is needed, for example in French: *"Les éléphants asiatiques ont des petites oreilles."* where "*asiatiques*" (masculine plural of "*asiatique*", "asian") correlates with "*éléphants* (masculine plural), and "*petites*" (feminine plural of "*petit*", "small") correlates with "*oreilles*" (feminine plural, "ears").

15.6.4 Parsing with Marked Ordered Grammars

Parsing with marked ordered grammars is easier than with "normal" ordered grammars, since here it is possible to always expand the leftmost non-terminal in the prediction for the rest of the input, and thus produce a leftmost derivation. We start with the CF start symbol as the initial prediction; we start a control string predictor based on the control grammar, and try to develop the prediction to match the input. If the leftmost non-terminal in the prediction is marked, we continue its production guided by the control string we predict from the control grammar. If the leftmost non-terminal is not marked, we stack the control string predictor for the leftmost marked non-terminal and start a new one, as we did for the start symbol. So the data structure in our parser will be a stack of control string predictors (in addition to the prediction), where each control string predictor consists of a prediction for the rest of the control tokens plus some backtrack administration.

We will now demonstrate the parsing of the input string **`aabbbbbbcc`** with the grammar from Figure 15.26. Depicting the full two-level parsing process would require a figure in the style of Figure 4.3, but it would be very big and less than

useful, since all branches but one would be dead ends. We will therefore show only the successful choices for the control string; these are indicated by a double arrow (⇒). All other moves — the application of the CF grammar rule, the matching of the correctly predicted input symbol, the creation of a new control string predictor — are forced, and will be indicated by single arrows (→), without comment. The predicted control stack is shown in curly braces {} right after the non-terminal to which it pertains.

We start with the prediction S→S{control}, since it is not marked; we then get:

```
S{control} ⇒ S{A3} ⇒ S{0A23} → a[S]{A23} →
[S]{A23} ⇒ [S]{0A223} → a[S]{A223} →
[S]{A223} ⇒ [S]{1223} → [T]{223} → B[T]{23}c →
B{control}[T]{23}c ⇒ B{445}[T]{23}c →
b[B]{45}[T]{23}c → [B]{45}[T]{23}c → b[B]{5}[T]{23}c
→ [B]{5}[T]{23}c → [b]{}[T]{23}c → b[T]{23}c →
[T]{23}c → B[T]{3}cc →
B{control}[T]{3}cc ⇒ B{445}[T]{3}cc → b[B]{45}[T]{3}cc
→ [B]{45}[T]{3}cc → b[B]{5}[T]{3}cc → [B]{5}[T]{3}cc
→ [b]{}[T]{3}cc → b[T]{3}cc → [T]{3}cc → [ε]{}cc →
cc → c → ε
```

We see that there are two more points where a new control string predictor is started.

As with the "normal" ordered grammars, the input is recognized by the CF grammar and a normal parse tree results, so semantics can be attached in the usual way.

Kulkarni and Shankar [279] give very efficient LL(1) and LR(1) parsers for marked ordered grammars.

15.7 Recognition Systems

As said before, non-Chomsky grammars find their origin in user objections to Type 0 and Type 1 grammars. Proponents of recognition systems take the criticism one step further and ask: "Why do we cling to a generative mechanism for the description of our languages, from which we then laboriously derive recognizers,[1] when almost all we ever do is recognizing text? Why don't we specify our languages directly by a recognizer?" Some people answer these two questions by "We shouldn't" and "We should", respectively.

Several recognition systems have been described over the years; examples are the analytic grammars by Gilbert [239], the TMGs by Birman and Ullman [246], and S/SL (Syntax/Semantics Language) by Barnard and Cordy [265, 50] (used in compiler construction). More modern, much more extensive recognition systems are PEGs (Parsing Expression Grammars) by Ford [286] and §-calculus by Jackson [291].

[1] People even write — and read — books about it.

15.7.1 Properties of a Recognition System

Basically a *recognition system* is a program for recognizing and possibly structuring a string. It can in principle be written in any programming language, but is in practice always written in a very specialized programming language, designed expressly for the purpose. Programs in such a programming language look like grammars, but their interpretation is profoundly different.

A very simple recognition expression is **a**, which recognizes the token **a** and passes over it; this means that it moves the read pointer in the input, which was in front of the **a**, to behind the **a**. If the expression is the entire program and **a** is the entire input, the program terminates successfully; what that means depends on the system, but it would be reasonable to assume that a data structure is made available, representing the recognized input, to which semantic operations can be applied. Likewise, the expression **a b** (or simply **ab**) recognizes **ab**, but if the input is **ac** it will fail and leave the read pointer before the **a**.

A more interesting expression is

a & ε

It recognizes an **a** but does not pass over it, so the read pointer remains where it was; the expression then continues to recognize the empty expression after the **&**, which of course succeeds. So the expression **a&ε** succeeds, but the end of the input is not reached, so if this is the entire program, it fails. More generally, an expression of the form

$$e_1 \ \& \ \cdots \ \& \ e_n$$

succeeds if all expressions $e_1, \cdots, e_n$ are recognized starting from the present read pointer, and it leaves the read pointer where e_n left it. If one of the expressions $e_1, \cdots, e_n$ fails, the whole expression fails and the read pointer is not moved.

In addition to these AND expressions there are OR expressions, which look like normal grammar alternatives, but behave differently. The expression

a / b / c

recognizes an **a**, a **b** or a **c**, but unlike its counterpart in a grammar it is "prioritized": the choices are tried in order and the first one to succeed wins. We shall see that this difference in interpretation with context-free grammars has several far-reaching consequences.

The expression **ab/a** will preferably recognize **ab** but will settle for **a** if there is no **ab**. On the other hand the expression **a/ab** will only recognize an **a**; if the input is **ab**, the first alternative still takes priority. This means that the second alternative in **a/ab** is useless, but unlike context-free grammars there is no algorithm to clean up a recognition program: it can be proved (Ford [286]) that it is in general undecidable whether an expression is useful (that is, whether it will ever match anything).

Another consequence of the "first matching alternative wins" principle is that a given expression E starting at a given position in the input recognizes and passes over exactly one segment of the input (or fails): there is no ambiguity and the matching

is unique. The recognition algorithm is fully deterministic. We shall see that this is a great help in designing an efficient (in fact linear-time) recognition algorithm.

Recognition expressions can be named, and the names can be used in other expressions or even in the same expression:

```
P  <-  ( P ) / [ P ] / ε
```

defines an expression **P** which recognizes correctly nested sequences of round and square parentheses. **P** can then be used in another recognition expression, for example

```
( & P
```

which recognizes the same strings as **P** except that the strings have to start with a **(**. This is a restriction that would be hard to express in a context-free grammar; see Problem 15.14.

A further consequence of the unique matching is that left-recursive rules are useless. For a rule like $A \leftarrow A\alpha$ to recognize a segment of the input both A and $A\alpha$ would have to match that segment. Since A can match only one segment due to the unique matching, this means that α must be ε, which turns the rule into $A \leftarrow A$, which just says that we have recognized an A when we have recognized an A. The recognition system PEG, for example, forbids left-recursive rules; unlike usefulness, left recursion can be tested.

As in EBNF, entities can be repeated by following them by a superscript asterisk: the expression **a*** recognizes zero or more **a**s. Actually, it does more than that: since it is equivalent to a call of **A** where **A** is defined by

```
A  <-  a A / ε
```

it recognizes the longest possible sequence of **a**s. The reason is that the first alternative of **A** continues to succeed as long as there are **a**s left. As with the alternatives above, repetitions like **a*** will match only one segment of the input: the longest one. A consequence of this is that the expression **a*****a** does not match any string: the **a*** moves over all **a**s present and at the end there will be no **a** left to match the trailing **a** in the expression.

Many recognition systems, including PEG, feature negation: the expression **!P** fails and recognizes nothing if **P** succeeds at this input position, and it succeeds and recognizes nothing if **P** fails at this point. Negation is generally useful but especially so in writing lexical analyzers:

```
\n / \t / !\\ .
```

recognizes **\n**, **\t** and any character except the backslash. (In a convention taken from lexical analyzers, the dot (**.**) at the end of the expression matches any character.)

Some programming languages have complicated conventions for comments. An example is the nesting comment in Pascal. In its basic form it consists of an opener, **(*** , some text, and a closer, ***)**. This construct is already non-trivial to recognize

since the text may contain `*`s and `)`s, just not the sequence `*)`. The matter is complicated, however, by the fact that the text may again contain comments; this is useful for commenting out code segments that already contain comments. So

```
(* i := -1; (* should actually start at 0 *) *)
```

is a correct comment in Pascal. Such comments are recognized by the expression

```
          Comment  ←  (* CommentElement* *)
CommentElement  ←  Comment / !*) .
```

which matches fairly well the description in the Pascal Manual. The idea is that a **`CommentElement`** is either a complete **`Comment`** or any character (`.`) provided we are not looking at the string `*)`. Ford [286] shows that PEG is quite suitable for integrating lexical analysis and parsing.

At the beginning of this chapter (page 475) we pointed out that the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ is the intersection of two CF languages, $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m$ and $\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n$, and we exploit that fact in the recognition program in Figure 15.28. A call of **`S`** first recognizes

```
S  ←  A c* !. & a* C
A  ←  a A b / ε
C  ←  b C c / ε
```

Fig. 15.28. A recognition program for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

the string $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^m$, makes sure there are no left-over characters, backtracks over the string, and then recognizes $\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n$, which only succeeds if $m = n$. This shows that recognition systems can handle languages that are not CF.

15.7.2 Implementing a Recognition System

One of the best and most surprising features of recognition systems is that they can be converted relatively easily into linear-time recognizers. The first step is to identify all subexpressions in the recognition program. What is exactly a subexpression depends on the details of the algorithm, but for the recognition program in Figure 15.28 we can identify the following 12 subexpressions:

`S`, **`Ac`*****`!.`**, **`a`*****`C`**, **`A`**, **`C`**, **`a`***, **`c`***, **`a`**, **`b`**, **`c`**, **`.`**, and ε

Slightly different algorithms might also require subexpressions like **`Ac`*** or **`c`*****`!.`**, but for our explanation the above set suffices.

We now construct a recognition table T, very similar to the one in tabular parsing explained in Section 4.3. The horizontal axis is labeled with the terminal symbols in the input string and the vertical axis is labeled with the subexpressions identified above. The entry $T_{e,i}$ contains the length of the input segment the subexpression e recognizes at position i; since an expression can recognize only one segment in a given position, we can be sure that an entry of T can never contain more than one

length. This makes the process of filling the table much easier. If the subexpression e does not recognize a segment in position i, the entry $T_{e,i}$ is empty.

As in Section 4.3 there are two ways to fill in the recognition table: top-down and bottom-up. In the top-down approach we start by trying to find out what length of input is recognized by **S** from position 1. This test is implemented as a call of a routine for **S** with 1 as the position parameter. By the time the length has been determined and the call returns, the answer is stored in $T_{S,1}$; it is either one integer or "no". In addition, all intermediate answers obtained in computing $T_{S,1}$ are stored as well, so no entry is computed more than once.

Since recognition systems cannot contain left-recursive rules, we are safe from loops caused by the computation of $T_{A,k}$ resulting in another call of $T_{A,k}$, and so the sequence of calls will terminate properly.

The result of recognizing the string **aaabbbccc** with the program of Figure 15.28 is shown in Figure 15.29. For example, the entry $T_{\mathtt{a}^*\mathtt{C},1}$ has a 9 because $T_{\mathtt{a}^*,1}$

S	9									
Ac*!.	9									
a*C	9									
A	6	4	2	0						
C				6	4	2	0			
a*	3	2	1	0						
c*							3	2	1	0
a	1	1	1	-						
b				1	1	1	-			
c							1	1	1	-
.										-
ε				0			0			0
Input:	**a**	**a**	**a**	**b**	**b**	**b**	**c**	**c**	**c**	**#**
Position:	1	2	3	4	5	6	7	8	9	10

Fig. 15.29. Packrat parsing of the string **aaabbbccc**

has a 3 and $T_{C,4}$ has a 6. The entries marked **-** are entries where the recursive descent has found the absence of a match; the empty entries are never touched by the algorithm. This algorithm is called *packrat parsing* by Ford [284], because, like a packrat, it stores and remembers everything it has ever seen.

In the bottom-up method, the entries are filled starting at the bottom right corner, working upwards through the columns and working leftwards from column to column (Birman and Ullman [246]). The top left element is the last to be filled in, and

concludes the recognition of the input string. See Figure 15.30, which shows that the bottom-up method computes many more values.

Subexpressions that have parts which recognize the empty string are handled correctly, since the empty string recognition is available as an entry with value 0. For example, an **a*****C** of length 6 is recognized correctly in position 4 because an **a*** of length 0 was recognized in position 4.

The order in which the subexpressions appear in the first index of T requires some care in the bottom-up method, since it would be wrong, for example, to compute $T_{\mathtt{a}^*,k}$ before $T_{\mathtt{a},1}$. In fact the subexpressions must be ordered so that the first member of a subexpression comes lower in the table than the subexpression itself. For example, **Ac*****!.** must come higher up in the table than **A**, as indeed it does in Figure 15.30. Since the values in the columns are computed from the bottom upwards, this causes

S	9	-	-	-	-	-	-	-	-	0
Ac***!.**	9	-	-	-	-	-	-	-	-	0
a***C**	9	8	7	6	4	2	0	0	0	0
A	6	4	2	0	0	0	0	0	0	0
C	0	0	0	6	4	2	0	0	0	0
a*	3	2	1	0	0	0	0	0	0	0
c*	0	0	0	0	0	0	3	2	1	0
a	1	1	1	-	-	-	-	-	-	-
b	-	-	-	1	1	1	-	-	-	-
c	-	-	-	-	-	-	1	1	1	-
.	1	1	1	1	1	1	1	1	1	-
ε	0	0	0	0	0	0	0	0	0	0
Input:	**a**	**a**	**a**	**b**	**b**	**b**	**c**	**c**	**c**	**#**
Position:	1	2	3	4	5	6	7	8	9	10

Fig. 15.30. Bottom-up tabular parsing of the string **aaabbbccc**

$T_{\mathtt{A},k}$ to be computed before $T_{\mathtt{Ac}^*\mathtt{!.},k}$. Such an ordering is always possible since the expressions cannot be left-recursive. Determining a correct order is pretty tedious and the top-down method is more efficient anyway; but see Problem 15.15.

It is interesting to note that the recognition system works without modification for input sequences that contain non-terminals, for example resulting from previous parsings.

15.7.3 Parsing with Recognition Systems

Although recognition systems are definitely non-Chomskian, top-down parsing with them results in recognition tables very similar to those in CYK parsing, and the techniques presented in Chapter 4 to extract parsings from them can be applied almost unchanged. We will discuss here the conversion to a parse-forest grammar. A few details have to be considered. Recognition systems usually have at least two constructs not occurring in CF grammars: the AND separator **&** and the negation **!**; also, the interpretation of the OR separator **/** differs slightly from that of the alternatives separator | in CF grammars.

The AND separator, for example in **P←A&B**, means that **P** is established by two parsings, both starting at the same input position, where the length of **P** is determined by the length of the last subexpression. Parse forest grammars have little trouble expressing this:

$$\begin{array}{rcl} \texttt{P\_}k\texttt{\_}l & \leftarrow & \texttt{A\_}k\texttt{\_}m\ \texttt{\&}\ \texttt{B\_}k\texttt{\_}l \\ \texttt{A\_}k\texttt{\_}m & \leftarrow & \cdots \\ \texttt{B\_}k\texttt{\_}l & \leftarrow & \cdots \end{array}$$

where m may be smaller than, equal to, or larger than l.

The negation **!A** succeeds at an input position k when **A** cannot be recognized there; it produces the rule

$$\texttt{!A\_}k\texttt{\_0} \ \leftarrow\ \varepsilon$$

It fails when **A** is recognized, and then no rule for it is produced. As a side effect, however, rules have been produced for the recognition of **A**. These rules are now unreachable and can be cleaned out in the usual way (Section 2.9.5).

The same side effect occurs in OR separators. Suppose we have a rule **R←AB/CD/EF**, and suppose **A** is found, **B** is not found and **C** and **D** are found. Then **R** is parsed as **CD**; this results in the rule $\texttt{R\_}k\texttt{\_}l\texttt{←C\_}k\texttt{\_}m\ \texttt{D\_}(k+m)\texttt{\_}(l-m)$. But in the process a rule $\texttt{A\_}k\texttt{\_}p\texttt{←}\cdots$ has been created, which is now unreachable.

Since the result of a recognition system is unambiguous, the process yields a parse-tree grammar rather than a parse forest grammar. The parse tree grammar resulting from the table in Figure 15.29 does not require clean-up. It is given in Figure 15.31.

15.7.4 Expressing Semantics in Recognition Systems

Since an almost normal parse tree results, semantics can be attached to it in the usual way. The two constructs that make it different from a CF parse tree are the repetition expression and the AND expression. Handling the semantics of the AND expression is straightforward:

```
P  ←  Q & R  {P.attr := func(Q.attr, R.attr);}
```

The semantics of the repetition allows two interpretations, as in Section 2.3.2.4, but the iterative one is the most natural here. **A*** could yield an attribute which is an array of the attributes of the separate **A**s.

```
    S_1_9  ←  Ac*!._1_9 & a*C_1_9
Ac*!._1_9  ←  A_1_6 c*_7_3 !._10_0
  a*C_1_9  ←  a*_1_3 C_4_6
    A_1_6  ←  a_1_1 A_2_4 b_6_1
    A_2_4  ←  a_2_1 A_3_2 b_5_1
    A_3_2  ←  a_3_1 A_4_0 b_4_1
    A_4_0  ←  ε
    C_4_6  ←  b_4_1 C_5_4 c_9_1
    C_5_4  ←  b_5_1 C_6_2 c_8_1
    C_6_2  ←  b_6_1 C_7_0 c_7_1
    C_7_0  ←  ε
   a*_1_3  ←  a_1_1 a*_2_2
   a*_2_2  ←  a_2_1 a*_3_1
   a*_3_1  ←  a_3_1 a*_4_0
   a*_4_0  ←  ε
   c*_7_3  ←  c_7_1 c*_8_2
   c*_8_2  ←  c_8_1 c*_9_1
   c*_9_1  ←  c_9_1 c*_10_0
  c*_10_0  ←  ε
  !._10_0  ←  ε
```

Fig. 15.31. Parse tree grammar from the packrat parsing in Figure 15.29

15.7.5 Error Handling in Recognition Systems

Without special measures, a recognition system behaves as badly on incorrect input as any backtracking top-down system: it rejects all hypotheses it generated, backtracks to the beginning of the input and says: "No". Special measures are, however, possible in this case. The idea is to have a failing attempt to recognize an expression return the reason why it failed, just as a successful attempt returns the length of the recognized segment (Grimm [287]). This is demonstrated in Figure 15.32, where the input has been changed to **aaabbccc**. The attempt to recognize **S** in position 1 results in a call to **Ac***, which leads to a call of **A** in position 1. The **a** is recognized, and so is the **A** of length 4 in position 2, but the attempt to combine these into an **A** at position 1 fails, because there is no **b** in position 6. This information is made into the result of the call to **A** and is passed upwards, to **S**, where an error message is derived from it.

There is one situation in which error information needs to be merged: when an OR expression fails. Suppose we have an expression

```
trailing_Z_option  ←  Z / !.
```

which recognizes a **Z** or end-of-file; suppose **trailing_Z_option** is called at input position k; and a **Z** is present at position k but contains a syntax error. Then **Z** will return with error information about a position $l > k$, and next **!.** will fail with error information about position k. Since the first is more informative, we should then let **trailing_Z_option** fail with **Z**'s error information about position l.

S	no **b** at pos. 6								
Ac*!.	no **b** at pos. 6								
a*C									
A	no **b** at pos. 6	4	2	0					
C									
a*									
c*									
a	1	1	1	no **a** at pos. 4					
b				1	1	no **b** at pos. 6			
c									
.									
ε									
Input:	**a**	**a**	**a**	**b**	**b**	**c**	**c**	**c**	**#**
Position:	1	2	3	4	5	6	7	8	9

Fig. 15.32. Bottom-up tabular parsing of the string **aaabbbccc**

15.8 Boolean Grammars

One way to obtain more than context-free power and not stray too much from context-free grammars is to extend them with the Boolean combination operators *negation* ($\neg A$, all strings not produced by A) and *intersection* ($A \cap B$, all strings produced both by A and B). This yields the *Boolean grammars* or *Boolean closure grammars*.

This approach was pioneered by Schuler [248] who in 1974 gave a Turing machine recognizing languages described by Boolean formulas over CF languages and showed how to use it to define a context-sensitive fragment of ALGOL 60. In 1991 Heilbrunner and Schmitz [267] gave an $O(n^3)$ recognizer for Boolean grammars, based on an adapted Earley parser. Recently (2004-2005), Okhotin [288, 290, 289] has described properties of Boolean grammars and given several parsers for them. Even better, the author [290] shows how to use them to enforce the context conditions in a simple C-like programming language, including checks for use of undefined identifiers, multiple definitions, and calling a function with the wrong number of parameters, all of that in about 4 pages.

15.8.1 Expressing Context Checks in Boolean Grammars

The principle of the paradigm is "A correct program is the intersection of a syntactically correct program with one or more context-enforcing languages." This requires

building context-enforcing languages; their nature depends on the kind of context conditions that need to be checked.

To demonstrate the paradigm we will use an abstract form of a very, very simple programming language. Programs consist of an open brace, a set of definitions containing one or more different identifiers, a semicolon, zero or more applications of the defined identifiers, and a close brace. Identifiers are one letter long. An example of a "program" is **`{ij;ijiiij}`**, which could be an abstract of the C-like block **`{int i,j;i=4;j=i*i;print i,j;}`**.

The context conditions are: no multiple identifiers in the definitions section, and no undefined identifiers after the semicolon. The main tool for checking context conditions on identifiers is a language that matches pairs of identifiers: the set $\{wcw\}$, where the two ws are the same identifier, and c is any sequence of tokens. An example is **`j;ijiiij`**. A Boolean grammar for this demo language is given in Figure 15.33.

```
Program_S → '(' Body ')'
Body → Definitions ';' Idf_Seq_Opt & No_Undefined_Idfs
Definitions → Idf_Seq & No_Multiple_Idfs
No_Undefined_Idfs →
        Idf_Seq ';' |
        No_Undefined_Idfs Idf & Last_Idf_Is_Defined
Last_Idf_Is_Defined →
        Head_and_Tail_Idfs_Match |
        Idf Last_Idf_Is_Defined
No_Multiple_Idfs → ¬ Multiple_Idfs
Multiple_Idfs →
        Idf_Seq_Opt Head_and_Tail_Idfs_Match Idf_Seq_Opt

Any_Letter → 'a' | ... | 'z'
Any_Char → Any_Letter | ';'
Any_Seq → Any_Seq Any_Char | ε
Idf → Any_Letter
Idf_Seq → Idf | Idf_Seq Idf
Idf_Seq_Opt → Idf_Seq | ε
Head_and_Tail_Idfs_Match →
        'a' Any_Seq 'a' | ... | 'z' Any_Seq 'z'
```

Fig. 15.33. A Boolean grammar for the context-checked specification of a very, very simple programming language

The first rule specifies the syntactic shape of a program. A **`Body`** has the form **`Definitions ';' Idf_Seq_Opt`** and has no undefined applied identifiers. A body with no undefined applied identifiers **`No_Undefined_Idfs`** is either a body with no applied identifiers at all, or an already checked body followed by an identifier and that last identifier is also defined. The last identifier of a string **`Idf`** $\cdots$ **`;`** $\cdots$ **`Idf`** (which is what we are looking at by now) is defined if the head and tail identifiers

match (because the first **Idf** is in the definition section), or we take away the first identifier and then a string in which the last identifier is defined remains. This makes sure all identifiers are defined.

Next, **Definitions** is a sequence of identifiers with no multiple copies in it. **No_Multiple_Idfs** is of course the negation of **Multiple_Idfs**. And a sequence of identifiers with multiple copies is any sequence which contains a subsequence the head and tail items of which match. Elementary, my dear Watson... The last few rules complete the full context-checked specification of the very simple programming language. Okhotin [290] gives a much more extensive example, based on similar principles.

When we compare our exposition to the above paper, we see that we have not only restricted ourselves to a simplification of a simplification of a programming language, but that we have also cut an enormous corner. Specifying the language **Head_and_Tail_Idfs_Match** for identifiers of unrestricted length is much more complicated than for identifiers of length 1, and requires substantial trickery, which is explained in Okhotin [283].

It should also be noted that negation in a production system has weird properties, and soon leads to paradoxes. The simple rule **S**→¬**S** describes the set of strings that it does not contain; in other words, a string is in **S** if it is not in **S**. Only slightly better is the grammar **S**→¬**T; T**→**T**. Since **T** produces nothing, **S** produces all strings, i.e. Σ^*. And **S**→**0**|**1**|¬**S**[**0**|**1**] contains any string Z over [**0**|**1**]* provided it does not contain the string obtained by removing the last token from Z; it is unclear what that means. For ways to catch and/or tame these paradoxes see Heilbrunner and Schmitz [267] and Okhotin [288].

15.8.2 Parsing with Boolean Grammars

Boolean grammars can be parsed using tabular parsing, in a way similar to recognition systems in Section 15.7.3; see Heilbrunner and Schmitz [267] and Okhotin [288]. Generalized LR and LL parsing is also possible (Okhotin [289]).

15.8.3 §-Calculus

§-calculus (Jackson [285, 291]) adds another feature to the non-Chomsky arsenal of Boolean grammars: the possibility to assign a recognized segment of the input plus its parse tree to a variable of type non-terminal and to use that variable further on in the rule.

For one thing, the definition of **head_and_tail_idfs_match** becomes trivial with this facility:

```
grammar head_and_tail_idfs_match {
    S ::= $x(Idf) Any_Sequence x;
};
```

Here the first identifier is read and assigned to the local variable **x**. Its value is retrieved at the end of the rule and used to parse the last identifier; that parsing of course only succeeds if the two identifiers are identical.

It is interesting to see how this feature is implemented. Conceptually, the moment the assignment is made, a new grammar rule is created in which the non-terminal variable is substituted out. For example, once the **arg2** has been recognized in the input segment **arg2){print(arg1+arg2**, a new rule

```
S ::= "arg2" Any_Sequence "arg2";
```

is created, which then enforces the head-tail match. Note that this makes non-terminal variable assignment in §-calculus similar to logic variable binding in Prolog.

In addition to local variables there are global variables, and functions to manipulate and use them. This opens up a gamut of possibilities that could fill a book, which is exactly what the inventor did [291], and we refer the reader to it.

For completeness, the §-grammar for $\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n$ is:

```
grammar AnBnCn {
    S ::= (('[abc]+')<A>)<B>;
    A ::= X 'c+';
    X ::= "a" [X] "b";
    B ::= 'a+' Y;
    Y ::= "b" [Y] "c";
};
```

Although §-grammars are defined as generating devices, they can equally well be seen as recognition devices. Jackson [291] describes a parser for them based on a push-down automaton the stack elements of which are a restricted form of trees; this implements the variable substitution mechanism. In addition, it uses many optimizations. The time complexity is unknown; in practice it is almost always less than $O(n^2)$ and always less than $O(n^3)$.

15.9 Conclusion

Three non-Chomsky systems stand out in the landscape at the moment: attribute grammars, as the most practical; VW grammars, as the most elegant and mathematically satisfying; and Boolean systems like PEG, Boolean grammars, and §-calculus, as the most promising. Coupled grammars are an interesting fourth.

Parsing of the non-Chomsky systems is generally based on top-down depth-first search, sometimes with some guidance against left recursion. Only TAGs are habitually parsed with a bottom-up method, and recognition systems can be conveniently handled with both methods.

Problems

Problem 15.1: Design a systematic way to write CS grammars for languages $t_1^n t_2^n \cdots t_k^n$ for any set of k symbols.

Problem 15.2: *a*. Write a VW grammar for the language *ww* where *w* is any string of **a**s and **b**s. *b*. Convert it to a working Prolog program. What is its time dependency?

Problem 15.3: How is negation (**where M not equals N**) expressed in a VW grammar? How about **where X symbol not equals Y symbol**?

Problem 15.4: Refer to the grammar in Figure 15.10. *a*. Explain how it produces **i symbol**, **i symbol**, **i symbol**, **ii symbol**, **ii symbol**, **ii symbol**, **iii symbol**, **iii symbol**, **iii symbol**. *b*. If **N** is chosen to be ε in the right hand side of the rule for $\mathtt{text}_s$, what does the grammar produce?

Problem 15.5: Design a TAG for the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n$. Hint: create an auxiliary tree that inserts **ab** between the **a** and the **b** in the elementary tree and at the same time inserts **cd** between the **c** and the **d**.

Problem 15.6: Design a TAG for the language *ww* where *w* is any string of **a**s and **b**s. Hint: It's all cross-dependencies.

Problem 15.7: If your native language is not Dutch or Swiss German, try to find examples of cross-dependencies in your native language. If you are Dutch or Swiss German, try to find examples of multiple cross-dependencies, for example of the type $A_1A_2A_3B_1B_2B_3$.

Problem 15.8: Expand the parsing algorithm sketched in Section 15.4.2 into a complete algorithm, and compare the result to Vijay-Shankar and Joshi's version [264].

Problem 15.9: Research Project: Devise a reasonable error reporting and recovery technique for TAGs.

Problem 15.10: Using the obvious coupled grammar for the language *ww* where *w* is any string of **a**s and **b**s, simulate the top-down parsing of the input string **aaaaaa**. Why does the system backtrack even on obvious parsings like **abbabb**?

Problem 15.11: The official definition of coupled grammars demands that when a grammar rule uses one component of a non-terminal, it has to use all components, and use them in the right order. There seems to be little reason to demand this. The full backtracking top-down parsing algorithm is not affected by it, for example. On the other hand, the gain from dropping it is not obvious either. Examine the consequences of lifting this restriction.

Problem 15.12: Take the rhetorical questions at the beginning of Section 15.7 at face value and give reasons why we should prefer grammars over recognition systems.

Problem 15.13: *a*. At the end of Section 15.7.1 we use the expression **!.** to check for the absence of spurious characters. Construct a string $\mathtt{a}^p\mathtt{b}^q\mathtt{c}^r$ not in $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ that would be recognized by **S** if we had left the check out. *b*. Give a simpler recognition expression for **S**.

Problem 15.14: Given a CF grammar **G** with a rule **P** and a token **a**, devise a technique to obtain a grammar rule **Q** in **G** which produces **a&P**, that is, all the strings which **P** produces that start with an **a**.

Problem 15.15: *a*. Draw up the precise criteria for the order of subexpressions in bottom-up PEG recognition. *b*. Design an algorithm for obtaining the desired order.

Problem 15.16: Design a way to derive a root set (Section 12.2.1.1) for a recognition system like PEG, and use it to a create the corresponding suffix parser.

Problem 15.17: *History of Recognition Systems:* Birman and Ullman [246, p. 21] give a very clever recognition program for the language a^{n^2}, sequences of *a*s whose lengths are square numbers. The program is specified in a formalism that differs considerably from the one discussed here. Rewrite the program in a more PEG-like formalism, and convince yourself that it works.

16

Error Handling

Until now, we have discussed parsing techniques while largely ignoring what happens when the input contains errors. In practice, however, the input often contains errors, the most common being typing errors and misconceptions, but we could also be dealing with a grammar that only roughly, not precisely, describes the input, for example in pattern matching. So the question arises how to deal with errors. A considerable amount of research has been done on this subject, far too much to discuss in one chapter. We will therefore limit our discussion to some of the more well-known error handling methods, and not pretend to cover the field; see (Web)Section 18.2.7 for references to more in-depth information.

16.1 Detection versus Recovery versus Correction

Usually, the least that is required of a parser is that it detects the occurrence of one or more errors in the input, that is, we require *error detection*. The least informative version of this is that the parser announces: "input contains syntax error(s)". We say that the input contains a *syntax error* when the input is not a sentence of the language described by the grammar. All parsers discussed in the previous chapters (except operator-precedence) are capable of detecting this situation without extensive modification. However, there are few circumstances in which this behavior is acceptable: when we have just typed a long sentence, or a complete computer program, and the parser only tells us that there is a syntax error somewhere, we will not be pleased at all, not only about the syntax error, but also about the quality of the parser or lack thereof.

The question as to where the error occurs is much more difficult to answer; in fact it is almost impossible. Although some parsers have the "correct-prefix property", which means that they detect an error at the first symbol in the input that results in a prefix that cannot start a sentence of the language, we cannot be sure that this indeed is the place in which the error occurs. It could very well be that there is an error somewhere before this symbol but that this is not a syntax error at that point. Thus there is a difference in the perception of an error between the parser and the user. In

the rest of this chapter, when we talk about errors, we mean syntax errors, as detected by the parser.

So what happens when input containing errors is offered to a parser with a good error detection capability? The parser might say: "Look, there is a syntax error at position so-and-so in the input, so I give up". For some applications, especially highly interactive ones, this may be satisfactory. For many, though, it is not: often, one would like to know about all syntax errors in the input, not just about the first one. If the parser is to detect further syntax errors in the input, it must be able to continue parsing (or at least recognizing) after the first error. It is probably not good enough to just throw away the offending symbol and continue. Somehow, the internal state of the parser must be adapted so that the parser can process the rest of the input. This adaptation of the internal state is called *error recovery*.

The purpose of error recovery can be summarized as follows:

- an attempt must be made to detect all syntax errors in the input;
- equally important, an attempt must be made to avoid *spurious* error messages. These are messages about errors that are not real errors in the input, but result from the continuation of the parser after an error with improper adaptation of its internal state.

Usually, a parser with an error recovery method can no longer deliver a parse tree if the input contains errors. This is sometimes a source of considerable trouble. In the presence of errors, the adaptation of the internal state can cause semantic actions associated with grammar rules to be executed in an order that is impossible for syntactically correct input, which sometimes leads to unexpected results. A simple solution to this problem is to ignore semantic actions as soon as a syntax error is detected, but this is not optimal and may not be acceptable. A better option is the use of a particular kind of error recovery method, an *error correction* method.

Error correction methods modify the input as read by the parser so that it becomes syntactically correct, usually by deleting, inserting, or changing symbols. Error correction methods will not always change the input into the input actually intended by the user, and they do not pretend that they can. Therefore, some authors prefer to call these methods *error repair* methods rather than error correction methods. The main advantage of error correction over other types of error recovery is that the parser still can produce a parse tree and that the semantic actions associated with the grammar rules are executed in an order that could also occur for some syntactically correct input. In fact, the actions only see syntactically correct input, sometimes produced by the user and sometimes by the error corrector.

In summary, error detection, error recovery, and error correction require increasing levels of heuristics. Error detection itself requires no heuristics: a parser detects an error, or it does not. Determining the place where the error occurs may require heuristics, however. Error recovery requires heuristics to adapt the internal parser state so that it can continue, and error correction requires heuristics to repair the input.

With error handling comes the obligation to provide good error messages. Unfortunately there is little research on this subject, and most of the pertinent publications

are of a reflective nature, for example Horning [296], Dwyer [307] and Brown [312]. Explicit algorithmic support is rare (Kantorowitz and Laor [316]). The only attempt at automating the production of error messages we know of is Jeffery [328] who supplies the parser generator with a long list of erroneous constructs with desired error messages. The parser can then associate each error message with the state into which the erroneous construct brings the parser.

16.2 Parsing Techniques and Error Detection

Let us first examine how good the parsing techniques discussed in this book are at detecting an error. We will see that some parsing techniques have the correct-prefix property while other parsing techniques only detect that the input contains an error but give no indication where the error occurs.

16.2.1 Error Detection in Non-Directional Parsing Methods

In Section 4.1 we saw that Unger's parsing method tries to find a partition of the input sentence that matches one of the right-hand sides of the start symbol. The only thing that we can be sure of in the case of one or more syntax errors is that we will find no such partition. For example, suppose we have the grammar of Figure 4.1, repeated in Figure 16.1, and input `x+`. Fitting the first right-hand side of **`Expr`** with

```
Expr_S  ->  Expr + Term | Term
 Term   ->  Term x Factor | Factor
Factor  ->  ( Expr ) | i
```

Fig. 16.1. A grammar describing simple arithmetic expressions

the input will not work, because the input only has two symbols. We will have to try the second right-hand side of **`Expr`**. Likewise, we will have to try the second right-hand side of **`Term`**, and then we will find that we cannot find an applicable right-hand side of **`Factor`**, because the first one requires at least three symbols, and the second one only one. So we know that there are one or more errors, but we do not know how many errors there are, nor where they occur. In a way, Unger's method is too well prepared for dealing with failures, because it expects any partition to fail.

For the CYK parser, the situation is similar. We will find that if the input contains errors, the start symbol will not be a member of the top element of the recognition table.

So, the unmodified non-directional parsing methods behave poorly on errors in the input. A method to improve that behavior by using dynamic programming is shown in Section 16.4.

16.2.2 Error Detection in Finite-State Automata

Finite-state automata are very good at detecting errors. Consider for example the deterministic automaton of Figure 5.12, repeated in Figure 16.2.

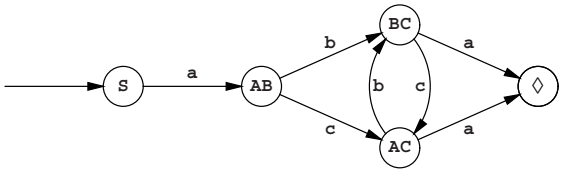


Fig. 16.2. Deterministic automaton for the grammar of Figure 5.6

When this automaton is offered the input **abcca**, it will detect an error when it is in state **AC**, on the second **c** in the input.

Finite-state automata have the correct-prefix property. In fact, they have the *immediate error detection property*, which we discussed in Chapter 8 and which means that an error is detected as soon as the erroneous symbol is first examined.

16.2.3 Error Detection in General Directional Top-Down Parsers

The breadth-first general directional top-down parser also has the correct-prefix property. It stops as soon as there are no predictions left to work with. Predictions are only dropped by failing match steps, and as long as there are predictions, the part of the input parsed so far is a prefix of some sentence of the language.

The depth-first general directional top-down parser does not have this property. It will backtrack until all right-hand sides of the start symbol have failed. However, it can easily be doctored so that it does have the correct-prefix property: the only thing that we must remember is the furthest point in the input that the parser has reached, a kind of high-water mark. The first error is found right after this point.

16.2.4 Error Detection in General Directional Bottom-Up Parsers

The picture is quite different for the general directional bottom-up parsers. They will just find that they cannot reduce the input to the start symbol. This is only to be expected because, in contrast to the top-down parsers, there is no test before an input symbol is shifted.

As soon as a top-down component is added, such as in Earley's parser, the parser regains the correct-prefix property. For example, if we use the Earley parser with the grammar from Figure 7.8 and input **a-+a**, we get the item sets of Figure 16.3 (compare this with Figure 7.11). We see that $itemset_3$ is empty, and the error is detected.

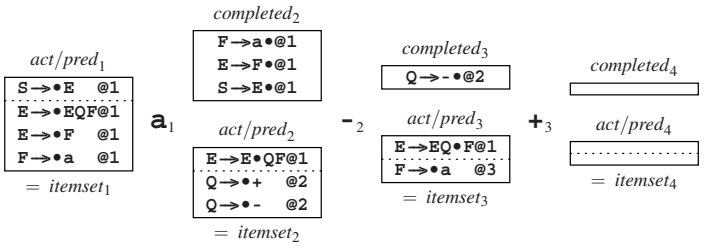


Fig. 16.3. Items set of the Earley parser working on `a-+a`

16.2.5 Error Detection in Deterministic Top-Down Parsers

In Sections 8.2.3 and 8.2.4 we have seen that strong-LL(1) parsers have the correct-prefix property but not the immediate error detection property, because in some circumstances they may make some ε-moves before detecting an error, and that full-LL(1) parsers have the immediate error detection property.

16.2.6 Error Detection in Deterministic Bottom-Up Parsers

Let us first examine the error detection capabilities of precedence parsers. We saw in Section 9.2.2 that operator-precedence parsers fail to detect some errors. When they do detect an error, it is because there is no precedence relation between the symbol on top of the parse stack and the next input symbol. This is called a *character-pair error*.

The other precedence parsers (simple, weak, extended, and bounded-right-context) have three error situations:

- there is no precedence relation between the symbol on top of the parse stack and the next input symbol (a *character-pair error*).
- the precedence relations indicate that a handle segment has been found and that a reduction must be applied, but there is no non-terminal with a right-hand side that matches the handle segment. This is called a *reduction error*.
- after a reduction has been made, there is no precedence relation between the symbol at the top of the stack (the symbol that was underneath the $\lessdot$) and the left-hand side to be pushed. This is called a *stackability error* or *stacking error*.

Reduction errors can be detected at an early stage by continuously checking that the symbols between the last $\lessdot$ and the top of the stack form the prefix of some right-hand side. Graham and Rhodes [295] show that this can be done quite efficiently.

In Section 9.6.3 we saw that an LR(1) parser has the immediate error detection property. LALR(1) and SLR(1) parsers do not have this property, but they do have the correct-prefix property. Error detection in GLR parsers depends on the underlying parsing technique.

16.3 Recovering from Errors

Error handling methods fall in different classes, depending on what level they approach the error. The general parsers usually apply an error handling method that considers the complete input. These methods use global context, and are therefore called *global error handling* methods. The Unger and CYK parsers need such a method, because they have no idea where the error occurred. These methods are very effective, but the penalty for this effectiveness is paid for in efficiency: they are very time consuming, requiring at least cubic time. As the general parsing methods already are time consuming anyway, this is usually deemed acceptable. We will discuss such a method in Section 16.4.

On the other hand, efficient parsers are used because they are efficient. For them, error handling methods are required that are less expensive. We will discuss the best known of these methods. They have the following information at their disposal:

- in the case of a bottom-up parser: the parse stack; in the case of a top-down parser: the prediction stack;
- the input string, and the point where the error was detected.

There are four classes of these methods: the *regional error handling* methods, which use some (regional) context around the point of error detection to determine how to proceed; the *local error handling* methods only use the parser state and the input symbol (local context) to determine what happens next; the *suffix methods*, which use zero context; and the *ad hoc* methods, which do not really form a class. Examples of these methods will be discussed in Sections 16.5, 16.6, 16.7 and 16.8.

In our discussions, we will use the terms *error detection point*, indicating the point where the parser detects the error, and *error symbol*, which indicates the input symbol on which the error is detected.

16.4 Global Error Handling

The most popular global error handling method is the *least-error correction* method. The purpose of this method is to derive a syntactically correct input from the supplied one using as few corrections as possible. Usually, a symbol deletion, a symbol insertion, and a symbol change all count as one correction (one edit operation).

It is important to realize that the number of corrections needed can easily be limited to a maximum: first, we compute the shortest sentence that can be generated from the grammar. Let us say it has length m. If the input has length n, we can change this input into the shortest sentence with a number of edit operations that is the maximum of m and n: change the first symbol of the input into the first symbol of the shortest sentence, etc. If the input is shorter than the shortest sentence, this results in a maximum of n changes, and we have to insert the last $m-n$ symbols of the shortest sentence. If the input is longer than the shortest sentence, we have to delete the last $n-m$ symbols of the input. This is not a very tight and useful maximum, but at least it shows the problem is finite. Also, when searching for a

least-error correction, if we already know that we can do it with, say, *k* corrections, we do not have to investigate possibilities known to require more.

With this in mind, let us see how such an error correction method works when incorporated in an Unger parser (Section 4.1). We will again use the grammar of Figure 16.1 as an example, again with input sentence `x+`. This is a very short sentence indeed, to limit the amount of work. The shortest sentence that can be generated from the grammar is `i`, of length one. The observation above limits the number of corrections needed to a maximum of two.

Now the first rule to be tried is **`Expr→Expr+Term`**. This leads to the following partitions:

`Expr`		**`+`**		**`Term`**		max:2
	?		1	`x+`	?	
	?	`x`	1	`+`	?	
	?	`x+`	1		?	
`x`	?		1	`+`	?	
`x`	?	`+`	0		?	
`x+`	?		1		?	cut-off

When we compare this table to tables like Figure 4.2, we note that it includes the number of corrections needed for each part of a partition in the right of the column; a question mark indicates that the number of corrections is yet unknown. The total number of corrections needed for a certain partition is the sum of the number of corrections needed for each of the parts. The top of the table also contains the maximum number of corrections allowed for the rule. For the parts matching a terminal, we can decide how many corrections are needed, which results in the column below the `+`. Also notice that we have to consider empty parts, although the grammar does not have ε-rules. The empty parts stand for insertions. The cut-off comes from the Unger parser detecting that the same problem is already being examined.

Now that it has this list of partitions, the Unger parser concentrates on the first partition in it, which requires it to derive ε from **`Expr`**. The partition already requires one correction, so the maximum number of corrections allowed is now one. The rule **`Expr→Expr+Term`** immediately results in a cut-off:

`Expr`	**`+`**	**`Term`**	max:1
?	1	?	cut-off

So we will have to try the other rule for **`Expr`**: **`Expr→Term`**. Likewise, **`Term→Term×Factor`** will result in a cut-off, so we will have to use **`Term→Factor`**. The rule **`Factor→(Expr)`** will again result in a cut-off, so **`Factor→i`** will be used:

Expr	max:1
Term	max:1
Factor	max:1
i	max:1
	1

So we find, not surprisingly, that input part ε can be corrected to **i**, requiring one correction (inserting **i**) to make it derivable from **Expr** (and **Term** and **Factor**).

To complete our work on the first partition of **x+** over the right-hand side **Expr+Term**, we have to examine if, and how, **Term** derives **x+**. We already need two corrections for this partition, so no more corrections are allowed because of the maximum of two. For the rule **Term→Term×Factor** we get the following partitions (in which we cheated a bit: we used some information computed earlier):

Term				**Factor**			
Term		**×**		**Factor**			max:0
	1		1	**x+**	?		too many corrections
	1	**×**	0	**+**	?		too many corrections
	1	**x+**	1		1		too many corrections
×	?		1	**+**	?		too many corrections
×	?	**+**	1		1		too many corrections
x+	?		1		1		cut-off

Each of these fails, so we try **Term→Factor**. The rule **Factor→(Expr)** then results in the following partitions:

Term					**Factor**				max:0
Term			**(**		**Expr**		**)**		max:0
	1			1		**x+**	2		too many corrections
	1		**×**	?		**+**	1		too many corrections
	1		**x+**	?			1		cut-off
×	1			1		**+**	1		too many corrections
×	1		**+**	?			1		too many corrections
x+	2			1			1		too many corrections

This does not work either. The rule **Factor→i** results in the following:

Term	max:0	
Factor	max:0	
i	max:0	
x+	2	too many corrections

So we get either a cut-off or too many corrections (or both). This means that the partition that we started with is the wrong one.

The other partitions are tried in a similar way, resulting in the following partition table, with completed error correction counts:

Expr		+		Term		max:2
	1		1	`x+`	>0	too many corrections
	1	`x`	1	`+`	1	too many corrections
	1	`x+`	1		1	too many corrections
`x`	1		1	`+`	1	too many corrections
`x`	1	`+`	0		1	
`x+`	?		1		1	cut-off

So, provided that we do not find better corrections later on, using the rule **`Expr→Expr+Term`** we find the corrected sentence **`i+i`**, by replacing the **`x`** with an **`i`**, and inserting an **`i`** at the end of the input.

Now the Unger parser proceeds by trying the rule **`Expr→Term`**. Continuing this process, we will find two more possibilities using two corrections: the input can be corrected to **`i×i`** by inserting an **`i`** in front of the input and replacing the **`+`** with another **`i`**, or the input can be corrected by replacing **`x`** with an **`i`** and deleting **`+`** (or deleting **`x`** and replacing **`+`** with an **`i`**).

This results in three possible corrections for the input, all three requiring two edit operations. Choosing between these corrections is up to the parser writer. If the parser is written to handle ambiguous input anyway, the parser might deliver three parse trees for the three different corrections. If the parser must deliver only one parse tree, it could just pick the first one found. Even in this case, however, the parser has to continue searching until it has exhausted all possibilities or it has found a correct parsing, because it is not until then that the parser knows if the input in fact did contain any errors.

As is probably clear by now, least-error correction does not come cheap, and it is therefore usually only applied in general parsers, because these do not come cheap anyway.

Lyon [294] has added least-error correction to the CYK parser and the Earley parser, although his CYK parser only handles replacement errors. In his version of the CYK parser, the non-terminals in the recognition table have an error count associated with it. In the bottom row, which is the one for the non-terminals deriving a single terminal symbol, all entries contain all non-terminals that derive a single terminal symbol. If the non-terminal derives the corresponding terminal symbol it has error count 0, otherwise it has error count 1 (a replacement). Now, when we find that a non-terminal A with rule $A \rightarrow BC$ is applicable, it is entered in the recognition table with an error count equal to the sum of that of B and C, but only if it is not already a member of the same recognition table entry, but with a lower error count.

Aho and Peterson [292] also added least-error correction to the Earley parser by extending the grammar with error productions, so that it produces any string of terminal symbols, with an error count. As in Lyon's method, the Earley items are extended with an error count indicating how many corrections were needed to create the item. An item is only added to an item set if it does not contain one like it which has a lower error count.

Tanaka and Fu [301] extended this method to context-sensitive parsers, in one of the few examples of error correction in systems stronger than context-free.

A completely different form of global error recovery is based on parsing by intersection and is treated in Section 13.5. It can give surprising results but there is hardly any research on it available yet.

16.5 Regional Error Handling

Regional error handling collects some context around the error detection point, consisting of a segment of the top of the stack and some prefix of the input, and reduces that part (including the error) to a left-hand side. Since it tries to collect a "phrase", which is a technical term for a terminal production of a non-terminal, this class of error handling methods is also often called *phrase level error handling*. Since the technique requires a reduction stack to participate in the desired reduction, it is applied exclusively to bottom-up parsers.

16.5.1 Backward/Forward Move Error Recovery

A well-known example of regional error handling in bottom-up parsers is the *backward/forward move* error recovery method, presented by Graham and Rhodes [295]. It consists of two stages: the first stage condenses the context around the error as much as possible. This is called the *condensation phase*. Then the second stage, the *correction phase*, changes the parsing stack and/or the input so that parsing can continue. The method is best applicable to simple precedence parsers, and we will use such a parser as an example.

Our example comes from the grammar and precedence table of Figure 9.9. Suppose that we have input **#n×+n#**. The simple precedence parser has the following parse stacks at the end of each step, up to the error detection point:

# ⋖	n × + n #	shift n
# ⋖ n ⋗	× + n #	reduce n
# ⋖ F ⋗	× + n #	reduce F
# ⋖ T ≐	× + n #	shift ×
# ⋖ T ≐ ×	+ n #	stuck

No precedence relation is found to exist between the × and the +, resulting in an error message that + is not expected.

Let us now examine the condensation phase in some detail. As said before, the purpose of this phase is to condense the context around the error as much as possible. The left-context is condensed by a so-called *backward move*: assuming a ⋗ relation between the top of the parse stack and the symbol on which the error is detected (that is, assuming that the parse stack built so far has the end of a handle as its top element), perform all possible reductions. In our example, no reductions are possible. Now assume a ≐ or a ⋖ between the top of the stack and the next symbol. This enables us to continue parsing a bit. This step is the so-called *forward move*: first we shift the next symbol, resulting in the following parse stack:

`# ⋖ T ≐ x ≐/⋖ +`	`n #` still stuck

Next, we disable the check that the top of the stack should represent a prefix of a right-hand side. Then, we continue parsing until either another error occurs or a reduction is called for that spans the error detection point. This gives us some right-context to work with, which can be condensed by a second backward move, if needed. For our example, this results in the following steps:

`# ⋖ T ≐ x ≐/⋖ + ⋖`	`n #` shift `n`
`# ⋖ T ≐ x ≐/⋖ + ⋖ n ⋗`	`#` reduce `n`
`# ⋖ T ≐ x ≐/⋖ + ⋖ F ⋗`	`#` reduce `F`
`# ⋖ T ≐ x ≐/⋖ + ⋖ T ⋗`	`#` reduce `T`
`# ⋖ T ≐ x ≐/⋖ + ≐ T' ⋗`	`#` proposed reduction includes error point

So now we have the situation depicted in Figure 16.4. This is where the correction

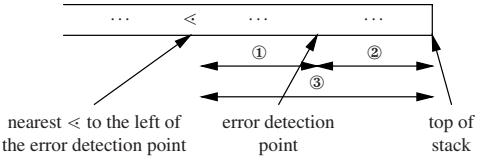
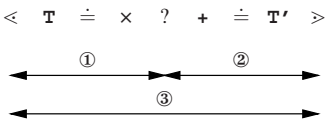


Fig. 16.4. Situation after the backward/forward moves

phase starts. The correction phase considers three parts of the stack for replacement with some right-hand side. These parts are indicated with ①, ② and ③ in Figure 16.4. Part ① is considered because the precedence at the error detection point could be ⋗, part ② is considered because the precedence at the error detection point could be ⋖, and part ③ is considered because this precedence could be ≐. Another option is to just delete one of these parts. This results in a fairly large number of possible changes, which now must be limited by making sure that the parser can continue after reducing the right-hand side to its corresponding left-hand side.

In the example, we have the following situation:



The left-hand sides that could replace part ① are: `E`, `T'`, `T`, and `F`. These are the non-terminals that have a precedence relation with the next symbol: the `+`. The only left-hand side that could replace part ② is `F`. Part ③ could be replaced by `E`, `T'`, `T`, and `F`. This still leaves a lot of choices, but some "corrections" are clearly better than others. Let us now see how we can discriminate between them.

Replacing part of the parse stack by a right-hand side can be seen as an edit operation on the stack. The cost of this edit operation can be assessed as follows. With every symbol, we can associate a certain insertion cost I and a certain deletion cost D. The cost for changing for example **T×** to **F** would then be $D(\mathtt{T})+D(\mathtt{\times})+I(\mathtt{F})$. These costs are determined by the parser writer. The cheapest parse stack correction is then chosen. If there is more than one with the same lowest cost, we just pick one.

Assigning identical costs to all edit operations, in our example, we end up with two possibilities, both replacing part ①: **T** (deleting the **×**), or **T×F** (inserting an **F**). Assigning higher costs to editing a non-terminal, which is not unreasonable, would only leave the first of these. Parsing then proceeds as follows:

# ⋖ T ≐ × ≐/⋖ + ≐ T′ ⋗	**#** error situation
# ⋖ T ≐ × ≐/⋖ + ≐ T′ ⋗	**#** correct error by deleting **×**
# ⋖ T ⋗ + ≐ T′ ⋗	**#** reduce **T**
# ⋖ T′ ⋗ + ≐ T′ ⋗	**#** reduce **T′**
# ⋖ E ≐ + ≐ T′ ⋗	**#** reduce **E+T′**
# ⋖ E ⋗	**#** reduce **E**
# ⋖ E′ ⋗	**#** reduce **E′**
# ⋖ S ⋗	**#** done

The principles of this method have also been applied in LR parsers. There, however, the backward move is omitted, because in an LR parser the state on top of the stack, together with the next input symbol, determine the reduction that can be applied. If the input symbol is erroneous, we have no way of knowing which reductions can be applied. For further details, see Pennello and DeRemer [300] and also Mickunas and Modry [299].

An interesting form of regional error handling is reported by Burke and Fisher [317]. Two parsers are used simultaneously, with one being several tokens ahead of the other; the input text between them is the region. This allows modifications to be made to the region when the first parser finds a syntax error. Several types of modifications can be applied, in such a way that the second parser never sees an error; see [317] for details. Charles [319] extends this technique with an impressive array of features, resulting in a robust error-correcting LALR parser.

16.5.2 Error Recovery with Bounded-Context Grammars

Error recovery, which is usually the most difficult part of error handling, is particularly easy when we use a bounded-context grammar (Section 9.3.1). The reason is that the bounded context allows the parser to get back on track quickly after an error, since little information is needed to start making correct decisions again.

A BRC parser using the grammar from Figure 9.2, the corresponding BC(2,1) parse table from Figure 9.10, and the input **#n×+n#**, performs the steps

# n $\gtrdot_{\mathtt{F}\to\mathtt{n}}$	**× + n #**	
# F $\gtrdot_{\mathtt{T}\to\mathtt{F}}$	**× + n #**	
# T $\lessdot$	**× + n #**	
# T × Error	**+ n #**	stuck

and finds that there is an Error relation between **T×** and **+**. Now, rather than trying to repair the situation at the gap, the parser tries to find the next context in which it *can* take a decision. To this end it has to shift at least 2 tokens; in this case that is enough to continue parsing:

# T × Error **+ n** $\gtrdot_{\mathtt{F \to n}}$	**#**	
# T × Error **+ F** $\gtrdot_{\mathtt{T \to F}}$	**#**	
# T × Error **+ T** $\gtrdot_{\mathtt{E \to E+T}}$	**#**	stuck

The parser detects that it cannot perform the indicated reduction, because rather than a **E** it finds a **×** on the stack. So seen from left to right the **+** is the error symbol and seen from right to left the **×** is the error symbol. The parser can now either delete a token or insert a token. If it deletes the **×** we get the context (**#T**,**+**) which is defined. If it deletes the **+** we get the context (**#T**,**×**) which shifts the **×** which leads to the context (**T×**,**T**) which is not defined. If it inserts, it can insert an **n** or a **(**. The first leads to a correct recovery, the second to failure. Assuming that the parser deletes the **×**,

# T	**+ T** $\gtrdot_{\mathtt{E \to E+T}}$ **#**	delete **×**

it continues as follows:

# T $\gtrdot_{\mathtt{E \to T}}$	**+ T** $\gtrdot_{\mathtt{E \to E+T}}$ **#**
# E $\lessdot$	**+ T** $\gtrdot_{\mathtt{E \to E+T}}$ **#**
# E + T $\gtrdot_{\mathtt{E \to E+T}}$	**#**
# E $\gtrdot_{\mathtt{S \to E}}$	**#**
# S Accept	**#**

Ruckert [322] describes the underlying algorithm; this integrated form of parsing and error recovery technique is called “robust parsing” in the paper, because the parser is not easily thrown off course. In [324] Ruckert shows that for the method to work the grammar must be a *continuous grammar*. A grammar is “continuous” if a small change in the input does not correspond to a discontinuous change in the parse tree, under a certain metric. It is shown that all BC grammars are continuous, but not vice versa, and that all continuous grammars are BCP but not vice versa. So we have for the grammars: BC $\subset$ continuous $\subset$ BCP.

16.6 Local Error Handling

All *local error recovery* techniques are so-called *acceptable-set error recovery* techniques. These techniques work as follows: when a parser detects an error, a certain set called the *acceptable-set* is computed from the parser state. Next, symbols from the input are skipped until a symbol is found that is a member of the acceptable-set. Then, the parser state is adapted so that the symbol that is not skipped becomes acceptable. There is a family of such techniques; the members of this family differ in the way they determine the acceptable-set, and in the way in which the parser state is adapted. We will now discuss several members of this family.

16.6.1 Panic Mode

Panic mode is probably the simplest error recovery method that is still somewhat effective. In this method, the acceptable-set is determined by the parser writer, and is fixed for the whole parsing process. The symbols in this set usually indicate the end of a syntactic construct, for example a statement in a programming language. For the programming language Pascal, this set could contain the symbols `;` and **`end`**. When an error is detected, symbols are skipped until a symbol is found that is a member of this set. Then, the parser must be brought into a state that makes this symbol acceptable. In an LL parser, this might require deleting the first few symbols of the prediction, in an LR parser this might involve popping states from the parse stack until a state is uncovered in which the symbol is acceptable.

The recovery capability of panic mode is often quite good, but many errors can go undetected, because sometimes large parts of the input are skipped. The method has the advantage that it is very easy to implement.

16.6.2 FOLLOW-Set Error Recovery

Another early acceptable-set recovery method is the *FOLLOW-set error recovery* method. The idea is applicable in an LL parser, and works as follows: when we are parsing a part of the input, and the top of the prediction stack results most recently from a prediction for the non-terminal A, and we detect an error, we skip symbols until we find a symbol that is a member of FOLLOW(A). Next, we remove the unprocessed part of the current right-hand side of A from the prediction, and continue parsing. As we cannot be sure that the current input symbol can follow A in the present context and is thus acceptable, this is not such a good idea. A better idea is to use that part of FOLLOW(A) that can follow A in this particular context, making sure that the symbol that is not skipped will be accepted, but this is not trivial to do.

The existence of this method is probably the reason that the family of acceptable-set error recovery methods is often called "FOLLOW-set error recovery". However, for most members of this family this is a confusing name.

A variant of this method that has become very popular in recursive descent parsers is based on the observation that at any point during the parsing process, there are a number of active non-terminals (for which we are now trying to match a right-hand side), and in general this number is larger than one. Therefore, we should use the union of the FOLLOW sets of these non-terminals, rather than the FOLLOW set of just the most recent of them. A better variant uses the union of those parts of the FOLLOW sets that can follow the non-terminals in this particular context. An expansion of this idea is the following: suppose the parser is in the following state when it detects an error:

$\cdots$	$a \cdots$
$\cdots$	$X_1 \cdots X_n$ #

We can then have the acceptable-set contain the symbols in FIRST(X_1), FIRST(X_2), $\cdots$, and #, and recover by skipping symbols until we meet a symbol of this acceptable-set, and then removing symbols from the prediction until the input symbol becomes acceptable.

Many variations of this technique exist; see for example Pemberton [304] and Stirling [314].

16.6.3 Acceptable-Sets Derived from Continuations

A very interesting and effective member of the acceptable-set recovery method family is the one discussed by Röhrich [305]. The idea is as follows. Suppose that a parser with the correct-prefix property detects an error in the input after having processed a prefix u. Because of the correct-prefix property, we know that this prefix u is the start of some sentence in the language. Therefore, there must be a *continuation*, which is a terminal string w, such that uw is a sentence of the language. Now suppose we can compute such a continuation. We can then correct the error as follows:

- Determine a continuation w of u.
- For all prefixes w' of w, compute the set of terminal symbols that would be accepted by the parser after it has parsed w', and take the union of these sets. If a is a member of this set, $uw'a$ is a prefix of some sentence in the language. This set is our acceptable-set. Note that it includes all terminal symbols in w, including the end marker.
- Skip symbols from the input until we find a symbol that is a member of this set. Note that as a result of this, everything up to the end marker may be skipped.
- Insert the shortest prefix of w that makes this symbol acceptable in front of this symbol. If everything up to the end marker was skipped, insert w itself.
- Produce an error message telling the user which symbols were skipped and which symbols were inserted.
- Restart the parser in the state where the error was detected and continue parsing, starting with the inserted symbols. Now the error is corrected, and the parser continues as if nothing has happened.

16.6.3.1 Continuation Grammars

Deriving acceptable sets from continuations requires a solution for two problems: how to determine the continuation and how to compute the acceptable-set without going through all possible parsings. Let us regard a grammar as a generating device. Suppose we are generating a sentence from a grammar, and have obtained a certain sentential form. Now, we want to produce a sentence from it as quickly as possible, using the fewest possible production steps. We can do this if we know for each nonterminal which right-hand side is the quickest "exit", that is, which right-hand side leads to a terminal production in as few production steps as possible.

We can compute these "quickest" right-hand sides in advance. To this end, we compute for each symbol the minimum number of production steps needed to obtain

a terminal derivation from it. We call this number the step count. Terminal symbols have step count 0; non-terminal symbols have an as yet unknown step count, which we set to infinity. Next, we examine each right-hand side in turn. If we already have a step count for each of the members of a right-hand side, the right-hand side itself needs the sum of these step counts, and the left-hand side needs one more if it uses this right-hand side. If this is less than we had for this non-terminal, we update its step count. We repeat this process until none of the step counts changes, as in a transitive closure algorithm.

If we started from a proper grammar, all of the step counts will now be finite. Now all we have to do is for each left-hand side to mark the right-hand side with the lowest step count. The grammar rules thus obtained are called a *continuation grammar*.

Let us see how this works with an example. Consider the grammar of Figure 8.9, repeated in Figure 16.5 for reference. The first pass over the right-hand sides shows

```
Sessions  →  Facts Question | ( Session ) Session
   Facts  →  Fact Facts | ε
    Fact  →  ! STRING
Question  →  ? STRING
```

Fig. 16.5. An example grammar

us that **Facts**, **Fact**, and **Question** each have step count 1. In the next pass, we find that **Session** has step count 3: its first alternative has two members with step count 1 each, plus 1 for the rule itself. The resulting continuation grammar is presented in Figure 16.6.

```
Sessions  →  Facts Question
   Facts  →  ε
    Fact  →  ! STRING
Question  →  ? STRING
```

Fig. 16.6. The continuation grammar of the grammar of Figure 16.5

16.6.3.2 Continuation in an LL Parser

In an LL parser, it now is easy to compute a continuation when an error occurs. We take the prediction, and derive a terminal string from it using only rules from the continuation grammar, processing the prediction from left to right. Each terminal that we meet ends up in the acceptable-set; in addition, every time a non-terminal is replaced by its right-hand side from the continuation grammar, we add to the acceptable-set the terminal symbols from the FIRST set of the current sentential form starting with this non-terminal.

Let us demonstrate this with an example. Suppose that we have the input **(? STRING ? STRING** for the LL(1) parser of Figure 8.10. When the parser detects an error, it is in the following state:

(? STRING	? STRING #
···	) Session #

Now a continuation will be computed, starting with the sentential form **) Session #**, using the continuation grammar. During this computation, when the prediction starts with a non-terminal, the FIRST set of the prediction will be computed and the non-terminal will be replaced by its right-hand side in the continuation grammar. The FIRST set is shown in square brackets below the line:

) Session # →

) [(!?] **Facts Question # →**

) [(!?] [!?] ε **Question # →**

) [(!?] [!?] [?] **? STRING #**

Consequently, the continuation is **) ? STRING #** and the acceptable-set contains **(**, **)**, **!**, **?**, **STRING** and **#**. We see that we should keep the **?** and insert the first symbol of the continuation, **)**. So the parser is restarted in the following state:

(? STRING	) ? STRING #
···	) Session #

and proceeds as usual.

16.6.3.3 Continuation in an LR Parser

Unlike an LL parser, an LR parser does not feature a sentential form which represents the rest of the input. It is therefore more difficult to compute a continuation. Röhrich [305] demonstrates that an LR parser can be generated that has a terminal symbol associated with each state of the handle recognizer so that we can obtain a continuation by pretending that the parser has this symbol as input when it is in the corresponding state. The sequence of states that the parser goes through when these symbols are given as input then determines the continuation. The acceptable-set consists of the terminal symbols on which a shift or reduce can take place (i.e. which are acceptable) in any of these states.

16.6.4 Insertion-Only Error Correction

Fischer, Milton and Quiring [303] propose an error correction method for LL(1) parsers using only insertions. This method has become known as the *FMQ* error correction method. In this method, the acceptable-set is the set of all terminal symbols.

Fischer, Milton and Quiring argue that the advantage of using only insertions (and thus no deletions or replacements) is that a syntactically correct input is built around the input supplied by the user, so none of the symbols supplied by the user are deleted or changed.

Not all languages allow insertion-only error correction. If, for example, all strings start with the token **`program`** and that token cannot occur anywhere else in the input, then an input with two **`program`** tokens in it cannot be corrected by insertion only. However, many languages allow insertion-only, and other languages are easily modified so that they do.

Let us investigate which properties a language must have for every error to be correctable by insertions only. Suppose we have an input $xa\cdots$ such that the start symbol does derive a sentence starting with x, but not a sentence starting with xa; so x is a correct prefix, but xa is not. Now, if this error is to be corrected by an insertion y, xya must again be a correct prefix. This leads to the notion of *insert-correctable* grammars: a grammar is said to be insert-correctable if for every prefix x of a sentence and every symbol a in the language there is a continuation of x that includes a (so an insertion can always be found). Fischer, Milton and Quiring demonstrate that it is decidable whether an LL(1) grammar is insert-correctable.

So, the FMQ error correction method is applicable in an LL(1) parser built from an insert-correctable grammar. In addition, the LL(1) parser must have the immediate error detection property. As we have seen in Section 8.2.4, the usual (strong-)LL(1) parser does not have this property, but the full-LL(1) parser does. Fischer, Tai and Milton [302] show that for the class of LL(1) grammars in which every non-terminal that derives ε does so explicitly through an ε-rule, the immediate error detection property can be retained while using strong-LL(1) tables.

Now, how does the error corrector work? Suppose that an error is detected on input symbol a, and the current prediction is $X_1 \cdots X_n$**#**. The state of the parser is then:

$\cdots$	$a \cdots$
$\cdots$	$X_1 \cdots X_n$ **#**

As a is an error, we know that it is not a member of FIRST($X_1 \cdots X_n$**#**). We also know that the grammar is insert-correctable, so $X_1 \cdots X_n$**#** must derive a terminal string containing a. The error corrector now determines the cheapest insertion after which a is acceptable. Again, every symbol has associated with it a certain insertion cost, determined by the parser writer; the cost of an insertion is the sum of the costs of the symbols in the insertion.

To compute the cheapest insertion, the error corrector uses some tables that are precomputed for the grammar at hand (by the parser generator). First, there is a table that we will call **`cheapest_derivation`**, giving the cheapest terminal derivation for each symbol (for a terminal, this is of course the terminal itself). Second, there is a table that we will call **`cheapest_insertion`** giving for each symbol/terminal combination (X, a) the cheapest insertion y such that $X \xrightarrow{*} ya\cdots$, if it exists, or an

indication that it does not exist. Note that in any prediction $X_1 \cdots X_n$**#** there must be at least one symbol X such that the (X, a) entry of the **cheapest_insertion** table contains an insertion (or else the grammar was not insert-correctable).

Going back to our parser, we can now compute the cheapest insertion z such that a becomes acceptable. Consulting **cheapest_insertion**(X_1, a), we can distinguish two cases:

- **cheapest_insertion**(X_1, a) contains an insertion y_1; in this case, we have found an insertion.
- **cheapest_insertion**(X_1, a) does not contains an insertion. In this case, we use **cheapest_derivation**(X_1) as the first part of the insertion, and continue with X_2 in exactly the same way as we did with X_1. In the end, this will result in an insertion $y_1 \cdots y_i$, where $y_1, \ldots, y_{i-1}$ come from the **cheapest_derivation** table, and y_i comes from the **cheapest_insertion** table.

The most serious disadvantage of the FMQ error corrector is that it behaves rather poorly on those errors that are better corrected by a deletion. Advantages are that it always works, can be generated automatically, and is simple.

Anderson and Backhouse [310] present a significant improvement of the implementation described above, which is based on the observation that it is sufficient to only compute the first symbol of the insertion: if we detect an error symbol a after having read prefix u, and $w = w_1 w_2 \cdots w_n$ is a cheapest insertion, then $w_2 \cdots w_n$ is a cheapest insertion for the error a after having read uw_1. So the **cheapest_derivation** and **cheapest_insertion** tables are not needed. Instead, tables are needed that are indexed similarly, but only contain the first symbol of the insertion. Such tables are much smaller, and easier to compute.

16.6.5 Locally Least-Cost Error Recovery

Like the FMQ error correction method, *locally least-cost error recovery* (see Backhouse [153] and Anderson et al. [311]) is a technique for recovering from syntax errors by editing the input string at the error detection point. The FMQ method corrects the error by inserting terminal symbols; the locally least-cost method corrects the error by either deleting the error symbol, or inserting a sequence of terminal or non-terminal symbols after which the error symbol becomes correct, or changing the error symbol. Unlike the least-error analysis discussed in Section 16.4, which considers the complete input string in determining the corrections to be made, the locally least-cost method only considers the error symbol itself and the symbol after that. The correction is determined by its cost: every symbol has a certain insertion cost, every terminal symbol has a certain deletion cost, and every replacement also has a certain cost. All these costs are determined by the parser writer. When considering if the error symbol is to be deleted, the cost of an insertion that would make the next input symbol acceptable is taken into account. The cheapest correction is then chosen.

This principle does not rely on a particular parsing method, although the implementation does. The method has successfully been implemented in LL, LR, and Earley parsers; see Backhouse [153], Anderson and Backhouse [306], Anderson et al. [311], and Choe and Chang [315] for details.

McKenzie et al. [321] extend this method by doing a breadth-first search over an ever deepening set of combinations of insertions and deletions. The correcting combinations are then "validated" in the order of cost, by applying them provisionally to the input and running the parser. If the parser accepts a predetermined number of tokens the correction is accepted; otherwise the original input is restored and the system proceeds to the next proposed correction.

Cerecke [325] limits the breadth-first search by analysing the LR automaton. Kim and Choe [326] incorporate the search for validations in the LR parse table.

Corchuelo et al. [327] take a very systematic approach to the problem. The operators "insert", "delete" and "validate" (called "forward move" in the paper) are introduced in the LR parsing mechanism on an equal footing with the usual "shift" and "reduce", in such a way that the original LR parse tables still suffice. This allows very pliable error recovery and easy implementation in an existing parser.

16.7 Non-Correcting Error Recovery

Although the error correction and error recovery methods discussed above have their good and bad points, they all have the following problems in common:

- On an error, they change the input and/or the parser state, using heuristics to choose one of the many possibilities. We can, however, never be sure that we picked the right change.
- Selecting the wrong change can cause an avalanche of spurious error messages. Only the least-error analysis of Section 16.4 does not have this problem.

A quite different approach to error recovery is that of Richter [313]. He proposes a method that does not have the problems mentioned above, but has some problems of its own. The author argues that we should not try to repair an error, because we cannot be sure that we get it right. Neither should we try to change parser state and/or input. The only thing that we can assume is that the rest of the input is a suffix (tail) of a sentence of the language. This is an assumption made in several error recovery methods, but the difference is that most error recovery methods assume more than that, in that they use (some of) the parser state information built so far.

16.7.1 Detection and Recovery

The error recovery method now works as follows: parsing starts with a parser for the original language, preferably one with the correct-prefix property. When an error is detected, it is reported, and the present parsing effort is abandoned. To analyze the remaining suffix, a parser derived from the suffix grammar, a so-called *suffix parser*, is started on it. The detected error symbol is not discarded: it could very well be a

correct beginning of a suffix, for example when the only actual error is a missing symbol.

If during the suffix scan another syntax error is detected, it is again reported, and the suffix parser is reset to its starting state, ready to accept another suffix. This guarantees that each reported error is a genuine syntax error, since the situation is found to be incompatible with the input from the previous error onwards, regardless of what came before. It is also different from and not caused by the previous error, since all information from before the previous one has been discarded. For the same reason no error is reported more than once. This maintains a high level of user confidence in the error messages, which is a great advantage. A possible disadvantage is, that in the presence of errors the parser is unable to deliver a meaningful parse tree.

Since the method does not correct existing input, it is called a *non-correcting error recovery* method.

When the method was first invented in 1985, it was hard to apply it in real-world parsers. It was easy enough to construct the suffix grammar (see Section 12.1), but that grammar was not amenable to the usual LL or LR methods, and general CF methods were too expensive — or at least deemed to be so. That changed with the invention of efficient, and sometimes even linear-time suffix parsers (Chapter 12). Another way of solving the problem is to use an efficient GLR or GLL parser (Chapter 11) and have it generate the suffix grammar implicitly on the fly. An example of this technique is described by van Deudekom and Kooiman [170]. Given the complexity of writing a linear suffix parser or an efficient GLR or GLL parser, the simplicity of a general CF parser, and the speed of present-day processors, it might be easier to use a general CF parser to do the suffix analysis; that approach has additional advantages, as we shall see in the next section.

16.7.2 Locating the Error

Although non-correcting error recovery cannot give spurious error messages, it can miss errors, even arbitrarily many of them. If our input is described by the grammar `S→(S)`, `S→[S]`, `S→`ε, which produces properly nested sequences of open and close parentheses and brackets, and the input is `(((()]]]`, the first `]` is detected as illegal and ends the correct prefix. But the rest, `]]]]` is a perfectly legal suffix, so only one error is reported. This is probably not a big disadvantage in an interactive environment.

When a directional parser finds a syntax error, the only thing we can say is that the error must have been somewhere in the input read so far. It can even be arbitrarily far back, at the start of the input. Suppose our input language consists of arithmetic expressions, and the input is *E*`)**3`, where *E* is a long, correct arithmetic expression and `**` is the exponentiation operator. The obvious error is that a left parenthesis was missing at the beginning: `(`*E*`)**3`.

In a non-correcting parser that uses a general CF parser for the suffix analysis we can do better. We can use the CF parser to scan the input text backwards, using the reverse grammar of the input language; that grammar probably has no redeeming

properties, but a general CF parser can handle it. If there is only one error, the backward scan will find an error at or to the left of the position of the forward error; the region between the two errors is called the *error interval*. In the example above the error is found at the start of the input, and the resulting error message

```
Syntax error:
unexpected ) at position N and
unexpected beginning of input at position 0
```

will give the user a good idea of where to look.

If there is more than one error, a similar scheme can be used to locate more errors, but care is required since error intervals may overlap. Richter [313] give the details.

16.8 Ad Hoc Methods

The *ad hoc error recovery* methods are called ad hoc because they cannot be automatically generated from the grammar. These methods are as good as the parser writer makes them, which in turn depends on how good the parser writer is in anticipating possible syntax errors. We will discuss three of these ad hoc methods: error productions, empty table slots and error tokens.

16.8.1 Error Productions

Error productions are grammar rules, added by the grammar writer so that anticipated syntax errors become part of the language (and thus are no longer syntax errors). These error productions usually have a semantic action associated with them that reports the error; this action is triggered when the error production is used. An example where an error production could be useful is the Pascal if-statement, which has the following syntax:

```
if-statement  →  IF boolean-expression
                 THEN statement else-part
   else-part  →  ELSE statement | ε
```

A common error is that an **if-statement** has an **else-part**, but the statement in front of the **else-part** is terminated by a semicolon. In Pascal, a semicolon is a statement separator rather than a statement terminator and is not allowed in front of an **ELSE**. This situation could be detected by changing the grammar rule for **else-part** into

```
else-part  →  ELSE statement | ε | ; ELSE statement
```

where the last right-hand side is the error production.

The most important disadvantages of error productions are:

- only anticipated errors can be handled;
- the modified grammar might (no longer) be suitable for the parsing method used, because conflicts could be introduced by the added rules.

The advantage is that a very adequate error message can be given. Error productions can be used profitably in conjunction with another error handling method, to handle some frequent errors on which the other method does not perform well.

16.8.2 Empty Table Slots

In most of the efficient parsing methods, the parser consults one or more parse tables and bases its next parsing decision on the result. These parsing tables have error entries (represented as the empty slots), and if one of these is consulted, an error is detected. In this error handling method, the empty table slots are used to refer to error handling routines. Each empty slot has its own error handling routine, which is called when the corresponding slot is consulted. The error handling routines themselves are written by the parser writer. By very careful design of these error handling routines, very good results can be obtained; see for example Conway and Wilcox [293]. In order to achieve good results, however, the parser writer must invest considerable effort. Usually, this is not considered worth the gain, in particular because good error handling can be generated automatically.

16.8.3 Error Tokens

Another popular error recovery method uses error tokens. An *error token* is a special token that is inserted in front of the error detection point. The parser will pop states from the parse stack until this token becomes valid, and then skip symbols from the input until an acceptable symbol is found. The parser writer extends the grammar with rules using this error token. An example of this is the following grammar:

```
      input  →  input input_line | ε
input_line   →  ERROR_TOKEN NEWLINE | STRING NEWLINE
```

This kind of grammar is often seen in interactive applications, where the input is line by line. Here, **ERROR_TOKEN** denotes the error token, and **NEWLINE** denotes an end of line marker. When an error occurs, states are popped until **ERROR_TOKEN** becomes acceptable, and then symbols are skipped until a **NEWLINE** is encountered.

This method can be quite effective, provided that care is taken in designing the rules using the error token.

16.9 Conclusion

In principle error handling is a hopeless task, in that the goal of having a computer handle errors properly in any intuitive meaning of the word is out of reach. In practice the far less lofty goal of not looping and not crashing is often already difficult to achieve. The techniques described in this chapter walk a middle ground: they define a metric for the corrections, thus creating an objective goal, and insert a token only when it can be proven that no looping can occur.

The techniques are very parse-method specific but often involve a search for the "best" correction; sometimes the results of this search can be precomputed.

Problems

Problem 16.1: 1. Can the error message learning system of Jeffery [328] (Section 16.1) be implemented in an strong-LL(1) parser? 2. Implement it in your favorite parser generator.

Problem 16.2: *Project*: The globally least-error correction method explained in Section 16.4 can give spectacularly wrong results; for example, if the input is text in a computer programming language and contains more than 2 errors, the input can sometimes be "corrected" by putting comment symbols around the entire text. This suggests that "most-recognized correction", in which the number of accepted tokens is maximized, may be preferred over "least-error correction". Apply this idea to an Unger, CYK or Earley parser and investigate.

Problem 16.3: *Research project*: Research error handling by intersection parsing.

Problem 16.4: In Section 16.5 we claim that regional error handling is applicable to bottom-up parsers only. Why can we not just apply the top-down counterparts of its actions to a top-down parser: predict as much as we can, and then try to match with insertions and deletions?

Problem 16.5: Give an intuitive argument why BC and BRC grammars allow the method of Section 16.5.2 to be applied, and BCP grammars do not, as stated at the end of that section.

Problem 16.6: In Section 16.6.3.2 the implementation of acceptable-set error recovery with continuations is described using a scan over the prediction, but if we are dealing with a recursive descent parser there is no explicit prediction. When the conceptual prediction is $X_1 \cdots X_n \#$, we are in the routine for X_1 and the other elements of the prediction are hidden in the calling stack. Devise a way to obtain the acceptable-set when needed without explicitly constructing the prediction.

Problem 16.7: In Section 16.6.4 the implementation of insertion-only error correction is described using a scan over the prediction, but if we are dealing with a recursive descent parser there is no explicit prediction, as in Problem 16.6. Devise a way to obtain the cheapest insertion when needed without explicitly constructing the prediction.

17

Practical Parser Writing and Usage

Practical parsing is concerned almost exclusively with context-free (Type 2) and regular (Type 3) grammars. Unrestricted (Type 0) and context-sensitive (Type 1) grammars are hardly used since, first, they are user-unfriendly in that it is next to impossible to construct a clear and readable Type 0 or Type 1 grammar and, second, all known parsers for them have exponential time requirements. Chapter 15 describes a number of polynomial-time and even linear-time parsers for non-Chomsky systems, but few have seen practical application. For more experimental results see (Web)Section 18.2.6.

Regular grammars are used mainly to describe patterns that have to be found in surrounding text. For this application a recognizer suffices. There is only one such recognizer: the finite-state automaton described in Section 5.3. Actual parsing with a regular grammar, when required, is generally done using techniques for CF grammars.

In view of the above we shall restrict ourselves to CF grammars in the rest of this chapter. We start with a comparative survey of the available parsing techniques (Section 17.1). Parsers can be interpretive, table-driven or compiled; the techniques are covered in Section 17.2.1. Section 17.3 presents a simple general context-free parser in Java, both for experimentation purposes and to show the nitty-gritty details of a complete parser. Parsers, both interpretive and compiled, must be written in a programming language; the influence of the programming language paradigm is discussed in Section 17.4. Finally, Section 17.5 exhibits a few unusual applications of the pattern recognition inherent in parsing.

17.1 A Comparative Survey

17.1.1 Considerations

The initial demands on a CF parsing technique are obvious: it should be general (i.e., able to handle all CF grammars), it should be fast (i.e., have linear time requirements) and preferably it should be easy to program. Practically the only way to obtain linear

time requirement is to use a deterministic method; there are non-deterministic methods which work in linear time (for example Bertsch and Nederhof [96]) but there is little practical experience with them.

There are two serious obstacles to this naive approach to choosing a parser. The first is that the automatic generation of a deterministic parser is possible only for a subset of the CF grammars. The second is that, although this subset is often described as "very large" (especially for LR(1) and LALR(1)), experience shows that a grammar that is designed to best describe the language without concern for parsing is virtually never in this set. It is true that for most reasonable grammars a slightly different grammar can be found that generates the same language and that does allow linear-time parsing, but here are two problems with this. Finding such a grammar almost always requires human intervention and cannot be automated. And using a modified grammar has the disadvantage that the resulting parse trees will differ to a certain extent from the ones implied by the original grammar. Furthermore, it is important to notice that no deterministic method can handle ambiguous grammars.

An immediate consequence of the above observations is that the stability of the grammar is an important datum. If the grammar is subject to continual revision, it is impossible or at least highly inconvenient to adapt each version by hand to the requirements of a deterministic method, and we have no choice but to use a general method. Likewise, if the grammar is ambiguous, we should use a general method.

If one has the luxury of being in a position to design the grammar oneself, the choice is simple: design the grammar to be LL(1) and use a predictive recursive descent parser. It can be generated automatically, with good error recovery, and allows semantic routines to be included in-line. This can be summarized as: parsing is a problem only if someone else is in charge of the grammar.

17.1.2 General Parsers

There are three general methods that should be considered: Unger's, Earley's and GLR.

17.1.2.1 Unger

An Unger parser (Section 4.1) is easy to program, especially the form given in Section 17.3.2, but its exponential time requirements limit its applicability to occasional use. The relatively small effort of adding a well-formed substring table (Section 17.3.4) can improve its efficiency dramatically, and in this form it can be very useful, especially if the average input string is limited to some tens of tokens. The thus modified Unger parser requires in principle a time proportional to n^{N+1}, where n is the number of tokens in the input and N is the maximum number of non-terminals in any right-hand side in the grammar, but in practice it is often much faster. An additional advantage of the Unger parser is that it can usually be readily understood by all participants in a project, which is something that can be said of almost no other parser.

17.1.2.2 Earley

A simple, robust and efficient version of the Earley parser has been presented by Graham, Harrison and Ruzzo [23]. It requires a time proportional to n^3 for ambiguous grammars (plus the time needed to enumerate the parse trees), at most n^2 for unambiguous grammars and n for grammars for which a deterministic method would work; in this sense the Earley parser is self-adapting. Since it does not require preprocessing on the grammar, it is possible to have one grammar-independent Earley parser and to supply it with the grammar and the input whenever a parsing is needed. If this is convenient, the Earley parser is preferable to GLR.

17.1.2.3 Generalized LR

At the expense of considerably more programming and some loss of convenience in use, a GLR parser (Section 11.1) will provide a parsing in slightly more than linear time for all but the most ambiguous grammars. Since it requires preprocessing on the grammar, it is convenient to generate a separate parser for each grammar (using a parser generator); if the grammar is, however, very unstable, the preprocessing can be done each time the parser is called. The GLR parser is presently the parser of choice for serious parsing in situations where a deterministic method cannot be applied and the grammar is reasonably stable.

As explained in Section 11.1, a GLR parser uses a table to restrict the breadth-first search and the question arises what type of table would be optimal. Lankhorst [166] determined experimentally that LR(0) and SLR(1) are about equally efficient, and that LALR(1) is about 5-10% faster; LR(1) is definitely worse. So, unless a speed-up of a few percent matters, the simple LR(0) is the recommended parse table.

17.1.2.4 Notes

It should be noted that if any of the general parsers performs in linear time, it may still be a factor of ten or so slower than a deterministic method, due to the much heavier administration they need.

None of the general parsers identifies with certainty a part of the parse tree before the whole parse tree is completed. Consequently, if semantic actions are connected to the grammar rules, none of these actions can be performed until the whole parse is finished. The actions certainly cannot influence the parsing process. They can, however, reject certain parse trees afterwards; this is useful to implement context conditions in a context-free parser.

17.1.3 General Substring Parsers

Although general substring parsing does not differ fundamentally from general full parsing (construct a substring grammar as explained in Section12.1 and use one of the above general CF parsers) a specific substring parser will be much more efficient. Rekers and Koorn [212] and Rekers [213, Chapter 4] describe the details of such a parser.

17.1.4 Linear-Time Parsers

Among the grammars that allow linear-time parsing, the operator-precedence grammars (see Section 9.2.2) occupy a special place, in that they can be ambiguous. They escape the general rule that ambiguous grammars cannot be parsed in linear time by virtue of the fact that they do not provide a full parse tree but rather a parse skeleton. If every sentence in the generated language has only one parse skeleton, the grammar can be operator-precedence. Operator-precedence is by far the simplest practical method; if the parsing problem can be brought into a form that allows an operator-precedence grammar (and that is possible for almost all formula-like inputs), a parser can be constructed by hand in a very short time.

17.1.4.1 Requirements

Now we come to the full linear-time methods. As mentioned above, grammars are not normally in a form that allows deterministic parsing and have to be modified by hand to be so. This implies that for the use of a deterministic parser at least the following conditions must be fulfilled:

- the grammar must be relatively stable, so that the modification process will not have to be repeated too often;
- the user must be willing to accept a slightly different parse tree than would correspond to the original grammar.

Speed is not an issue: any not inordinately long input parses in a fraction of a second with a deterministic parser on a modern machine.

It should again be pointed out that the transformation of the grammar cannot, in general, be performed by a program (if it could, we would have a stronger parsing method).

Leo's improvement of the Earley parser [32] may be a viable alternative. It requires linear time on all deterministic and many other grammars, and does not require preprocessing. Since it is interpreted, we expect it to lose perhaps two orders of magnitude in speed over a table-driven LR parser, but that might not be a problem on present-day machines. Experience with this type of parser is lacking, though.

17.1.4.2 Strong-LL(1) versus LALR(1)

For two deterministic methods, "strong-LL(1)"[1] (Section 8.2.2) and LALR(1) (Section 9.7), parser generators are readily available, both as commercial products and in the public domain. Using one of them will in almost all cases be more practical and efficient than writing your own; for one thing, while writing a parser generator may be (is!) interesting, doing a reasonable job on the error recovery is a protracted affair, not to be taken on lightly. So the choice is between (strong-)LL(1) and

[1] What is advertised as an "LL(1) parser generator" is almost always actually a strong-LL(1) parser generator.

LALR(1). Full-LL(1) or LR(1) are occasionally preferable, and some parser generators for these are available if needed. The main differences between (strong-)LL(1) and LALR(1) can be summarized as follows:

- LL(1) usually requires larger modifications to be made to the grammar than LALR(1).
- LL(1) allows semantic actions to be performed even before the start of an alternative; LALR(1) performs semantic actions only at the end of an alternative.
- LL(1) parsers are often easier to understand and modify.
- If an LL(1) parser is implemented as a recursive-descent parser, the semantic actions can use named variables and attributes, much as in a programming language. No such use is possible in a table-driven parser.
- Both methods are roughly equivalent as to speed and memory requirements; a good implementation of either will outperform a mediocre implementation of the other.

The difference between the two methods disappears largely when the parser yields a complete parse tree and the semantic actions are deferred to a later stage. In such a setup LALR(1) is obviously to be preferred.

People evaluate the difference in power between LL(1) and LALR(1) differently; for some the requirements made by LL(1) are totally unacceptable, others consider them a minor inconvenience, largely offset by the advantages of the method.

If one is in a position to design the grammar along with the parser, there is little doubt that LL(1) is to be preferred: not only will parsing and performing semantic actions be easier, text that conforms to an LL(1) grammar is also clearer to the human reader. A good example is the design of Modula-2 by Wirth (see *Programming in Modula-2 (Third, corrected edition)* by Niklaus Wirth, Springer-Verlag, Berlin, 1985).

17.1.4.3 Table Size

The table size of a deterministic parser is moderate, from 10K to 100K bytes for the deterministic parsers to megabytes for the non-canonical ones, and will be a problem in very few applications. The strongest linear-time method with negligible table size is weak precedence with precedence functions.

17.1.5 Linear-Time Substring Parsers

On the subject of practical linear-time substring parsers there is little difficulty; Bates and Lavie's [214] is the prime candidate. It is powerful, efficient and not too difficult to implement.

17.1.6 Obtaining and Using a Parser Generator

The approach is simple: search the Internet. There are a surprising number of parser generators out there, including some for general CF parsing, based both on Earley

and on GLR. We shall not give names or URLs here, since, as we said in the Preface, such information is ephemeral and the best URL is a few well-chosen search terms submitted to a good Web search engine.

For advanced handling of the parse tree there are a number of parse tree processor generator tools and prettyprinters for parse trees. Converting the parse tree to XML and using a Web browser to view it is a simple but viable alternative for the latter.

17.2 Parser Construction

Parsing starts with a grammar and an input string (A) supplied by the user and ends with the result (B) desired by the user. Even disregarding the actual parsing technique used, there are several ways to get from A to B. Also, actually there is more than one destination B: generally the user will want a structuring of the input, in the form of a parse tree or a parse grammar, to be processed further, but sometimes it is the semantics of the input that is desired immediately. An example is a parser for arithmetic expressions like `4+5*6+8` which is expected to produce the answer `42` directly. Another is a PostScript or HTML interpreter, where most of the input file is executed while it is being parsed. See Clark [355] for more about the difference between building a parse tree and having immediate semantics.

17.2.1 Interpretive, Table-Based, and Compiled Parsers

Just as a program in a programming language like C or Java can either be interpreted or compiled into a binary executable, a grammar can either be interpreted or compiled into a parser. But we have to be careful of what we mean by these words. Actually it is the combination of program and input which is interpreted, and it is the program only that is compiled into executable code. Likewise the combination of grammar and input can be interpreted (as shown in Figure 17.1), and the grammar on its own can be compiled into a parser by a parser generator (Figure 17.2). In both pictures the boxes marked with a 'U' in the top left corner represent files supplied by the parser user; those with 'P' are created by the parser writer; those with a $\times$ are executable programs; and unmarked boxes represent generated files.

Figure 17.1 is simple: the source code of the interpreter is fed through a compiler, which produces the actual interpreter. It is then given both the user grammar and ditto input, and does its work. Interpreters read the grammar every time a new input is offered; this has the advantage that always the most up-to-date version of the grammar is used. They are also often easier to write than parser generators. Interpreters have the disadvantage of being slow, both because of the overhead inherent in interpreting, and because the grammar is processed time and again. It is sometimes convenient to incorporate the grammar as static data in the interpreter, thus creating a parser specific to that grammar.

Figure 17.2, concerning compiled parsers, is more complicated. First the code of the parser generator is compiled into the parser generator proper; this happens out of sight of the user. The user then supplies the grammar to the parser generator, which

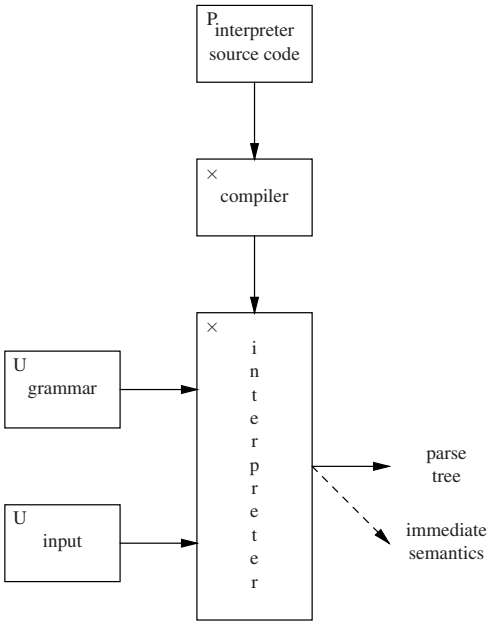


Fig. 17.1. Use of an interpreting parser (U = user-supplied; P = parser writer supplied)

turns it either into parse tables or into parser code, depending on the nature of the parser generator. The result is then compiled into the parser proper; if the parser is table-driven, a driver is added. The parser can then be used with different inputs as often as needed.

There is unfortunately no standard terminology to distinguish the three types; we shall call them "interpreters" (or "interpreting parsers"), "table-driven parsers", and "compiled parsers", respectively.

17.2.2 Parsing Methods and Implementations

If the parser is complicated it is usually easier to write it as an interpreter rather than to generate code for it. So most of the general CF parsers are programmed as interpreters and the situation in Figure 17.1 applies. Since general CF parsing is usually done in fairly experimental circumstances, in which the grammar is equally likely to change as the input, this has the additional advantage that one is not penalized for changing the grammar.

Compiling a general CF parser into code is not impossible, however; Aycock and Horspool [37] present a compiled Earley parser.

LL(1) parsers come in two varieties: compiled, using recursive descent (Section 8.2.6), in which a procedure is created for each non-terminal; and table-driven, using a table like the one in Figure 8.10 and a pushdown automaton as described in Section 6.2. The recursive descent version is probably more usual.

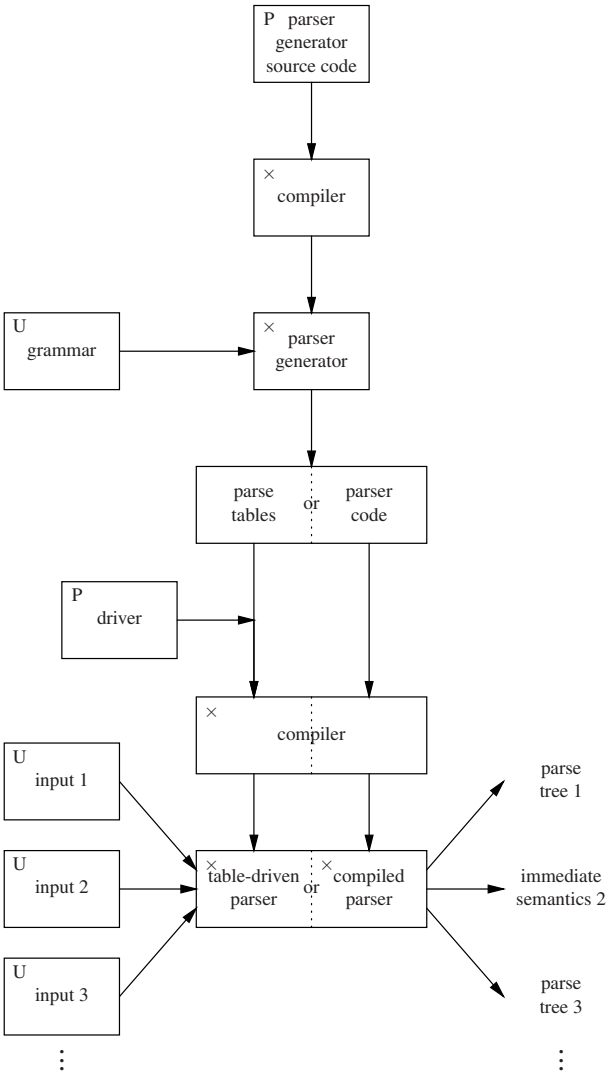


Fig. 17.2. Use of a compiled parser

Almost all LR parsers are table-driven, using tables like the one in Figure 9.28, and a pushdown automaton. Examples of compiled LR parsers are discussed by Pennello [70], Horspool and Whitney [85] and Bhamidipaty and Proebsting [354]. The recursive ascent parsers (Section 9.10.6) also lead to compiled LR parsers.

There is ample evidence that compiled parsers are faster than table-driven parsers, in addition to having the advantage that semantic routines can be included conveniently; see, for example, Waite and Carter [339], and the above papers. At least two causes for this phenomenon have been identified: we avoid the time re-

quired to identify the action to be performed and to prepare for calling it; and the often superb optimization done by present-day compilers.

An interesting possibility is to start the parser as an interpreter but to memoize the table information from every parsing decision made and reuse it when the same decision comes up again. This is known as *lazy table construction*. It is especially effective for lexical analysers, but can also help in modularizing the parsing process (Koskimies [342]). It has the possible disadvantage that only the activated part of the grammar gets checked.

For finite-state automata table-driven implementations are the norm. But see Jones [146] for compiled FS automata.

17.3 A Simple General Context-Free Parser

Although LL(1) and LALR(1) parsers are easy to come by, they are of limited use outside the restricted field of programming language processing. General CF parsers are available, but are often complicated, both to understand and to use. We will therefore present here in full detail a simple general parser that will yield all parsings of a sentence according to a CF grammar, with no restriction imposed on the grammar. It is small, written in Java, and enables the reader to experiment directly with a general CF parser that is under his or her full control. The parser described in the Sections 17.3.1 to 17.3.3 takes exponential time in the worst case; a memoization feature which reduces the time requirement to polynomial is discussed in Section 17.3.4. The interested reader who has access to a Prolog interpreter may wish to look into DCGs ("Definite Clause Grammars", Section 6.7). These may be more useful than the parser in this chapter since they allow context conditions to be applied, but they cannot handle left recursion unless special measures are taken, as for example those in Section 6.8).

17.3.1 Principles of the Parser

The parser, presented as a set of Java classes in Sections 17.3.2 to 17.3.4, is the simplest we can think of that puts no restrictions on the grammar. Since it searches a forest of possible parse trees to find the applicable ones, it is not completely trivial, though. The parser is an Unger parser in that it does a top-down analysis, dividing the input into segments that are to be matched to symbols in the pertinent right-hand sides. A depth-first search, using recursive descent, is used to enumerate all possibilities.

To avoid clutter, not all class declarations are shown in the printings of class declarations, and many declarations of administrative methods (straightforward constructors, `toString()`, etc.) are suppressed; the full working parser can be downloaded from this book's web site.

17.3.2 The Program

The central objects in the parser are **Goal**s, **RuleGoal**s, and **DottedGoal**s; they form a small linear hierarchy. A **Goal** contains a non-terminal **lhs**, a position in the input **pos**, and a length **length**; it represents an attempt to derive the indicated segment of the input from the non-terminal. **RuleGoal** is a subclass of **Goal**, in which the non-terminal has been narrowed down to a specific rule, **rule**. A **DottedGoal** is a **RuleGoal** with dots in the right hand side and the input segment; the positions of the dots are given as two integer fields **rhsUsed**, and **inputUsed**. A **DottedGoal** succeeds when it can match the remainder of the right-hand side to the remainder of the input segment.

The recursion in Java is used to search the space of all alternatives of non-terminals and of all possible segment lengths. The **DottedGoal**s generated by this search are put on a stack called **DottedGoalStack**. The **DottedGoalStack** starts off with a **DottedGoal** containing the start symbol and the entire input; whenever the stack becomes empty, a parsing has been found. To print a listing of the rules that led to the parsing, newly tried rules are stacked on a **RuleStack**, and removed from it when no more matchings for them can be found.[2]

The **DottedGoalStack** contains the active nodes in the parse tree, which is only a fraction of the nodes of the parse tree as already recognized (Figure 6.2). The leftmost derivation of the parse tree as far as recognized can be found on the stack **RuleStack**. When the **DottedGoalStack** becomes empty, a complete parsing has been found, recorded in the **RuleStack**.

The main class of the parser, shown in Figure 17.3 just loads the grammar from

```
public class TopDownParser {
    public static void main(String[] args) {
        String userGrammarName = "UserGrammar4";
        Grammar.load(userGrammarName);
        Grammar.parse();
    }
}
```

Fig. 17.3. The driver

a user class and parses the offered strings. The demo grammar file **UserGrammar4** specifies the grammar

```
S  ->  LSR | ε
L  ->  ( | ε
R  ->  )
```

[2] Somebody who would suggest that we are implementing a Prolog interpreter in disguise would be right.

This grammar forces the parser to match either an opening parenthesis or ε to each closing parenthesis, which makes most inputs very ambiguous. The demo applies it to the strings `())` and `(((())))`.

The class **`Grammar`** (Figure 17.4) uses a call to

```
import java.util.ArrayList;
public class Grammar {
    private static ArrayList<Rule> ruleList = new ArrayList<Rule>();
    public static Symbol startSym = null;
    public static Rule getRule(int n) {return ruleList.get(n);}
    public static void load(String filename) {
        ReadGrammar.readGrammar(filename);
    }
    public static void parse() {
        Input s;
        while ((s = ReadInput.readInput()) != null) TD.parse(s);
    }
}
```

Fig. 17.4. The class **`Grammar`**

`ReadGrammar.readGrammar(String filename)` (not shown) to read the grammar from a file and store its start symbol in **`startSym`** and its rules in the array list **`ruleList`**. The rules are objects of class **`Rule`** (not shown), each of which has two public fields, **`lhs`** and **`rhs`**. The tokens in the grammar and the input are objects of a class **`Symbol`** (not shown); this allows any lexical analyser to be plugged in independently.

The class **`Grammar`** also supplies the method **`parse()`** used above in **`TopDownParser`**. It reads a sequence of input strings and calls the top-down parser in class **`TD`** for each of them.

This brings us to the class **`TD`** (Figure 17.5), which supplies the methods **`parse(Input)`** and **`parsingFound()`**, and counts the number of derivations. The method **`parse(Input)`** starts by doing some initializations, which include creating a new **`RuleStack`** (not shown) and a new **`DottedGoalStack`** (not shown), and clearing the derivation count; for **`knownRuleGoals`** see the next paragraph. Next it prints the **`input`** in a message. Then the real parsing starts: **`parse(Input)`** creates a **`Goal`** consisting of the start symbol of the grammar, the start position 0, and the length of the input, and activates it by calling its method **`doWork()`**. When the activation returns, all parsings have been counted and reported. The method **`parsingFound()`** counts each parsing and reports it by printing the rule stack.

To prepare the way for a system to memoize known parsings, calls to methods in an object of class **`knownRuleGoals`** have already been placed in the presented code. We shall ignore them until Section 17.3.4.

The method **`doWork()`** in the class **`Goal`** (Figure 17.6) runs down the list

```
public class TD {
    static Input           input;
    static RuleStack       ruleStack;
    static DottedGoalStack dottedGoalStack;
    static KnownRuleGoals  knownRuleGoals;
    private static int     countDerivations;
    public static void parse(Input input) {
        TD.input = input;
        ruleStack = new RuleStack();
        dottedGoalStack = new DottedGoalStack();
        knownRuleGoals = new KnownRuleGoals();
        countDerivations = 0;
        System.out.println("Parsing \"" + input
                + "\" of length " + input.length());
        (new Goal(Grammar.startSym, 0, input.length())).doWork();
        System.out.println(countDerivations + " derivation"
                + (countDerivations == 1 ? "" : "s")
                + " found for string \"" + input + "\"\n");
    }
    public static void parsingFound() {
        countDerivations++;
        System.out.println("Parsing found:\n" + ruleStack.toString());
    }
}
```

Fig. 17.5. The class **TD**

```
public class Goal {
    Symbol lhs; int pos; int length;
    public void doWork() {
        for (int n = 0; n < Grammar.size(); n++) {
            Rule r = Grammar.getRule(n);
            if (r.lhs.equals(lhs))
                (new RuleGoal(this, r)).doWork();
        }
    }
}
```

Fig. 17.6. The class **Goal**

of grammar rules and for each rule **r** for the desired non-terminal it creates a **RuleGoal** containing the **Goal** and **r**, and activates it.

The method **doWork()** in the class **RuleGoal** (Figure 17.7) contains three calls to methods in **TD.knownRuleGoals**, which we ig-

```
public class RuleGoal extends Goal {
    Rule rule;
    public void doWork() {
        // avoid left recursion:
        if (TD.dottedGoalStack.contains(this)) return;
        // try to avoid rebuilding known parses:
        if (TD.knownRuleGoals.knownRuleGoalTable.containsKey(this)) {
            TD.knownRuleGoals.doWork(this); return;
        }
        TD.knownRuleGoals.startNewParsing(this);
        System.out.println("Trying rule goal " + toString());
        (new DottedGoal(this, 0, 0)).doWork();
    }
    public void doWorkAfterDone() {
        if (TD.dottedGoalStack.empty()) TD.parsingFound();
        else TD.dottedGoalStack.top().doWorkAfterMatch(length);
    }
}
```

Fig. 17.7. The class **RuleGoal**

nore until Section 17.3.4. For the moment we also ignore the call to **TD.dottedGoalStack.contains(this)**, which serves as a protection against problems with left recursion; it will be discussed in Section 17.3.3. So there is only one thing left to do for **RuleGoal.doWork()**: create a **DottedGoal** with both dots at the left end, and activate it. The method **doWorkAfterDone()** will be discussed later on.

The class **DottedGoal** (Figure 17.8) is the second-most complicated class of the parser, after the still mysterious class **KnownRuleGoals**. Its **doWork()** method is the first to show real action: it puts itself on the parsing stack **TD.dottedGoalStack**, puts the rule it contains on the rule stack **TD.ruleStack**, and leaves the upcoming intricate decisions to the private method **doAsTopOfStack()**.

This method has to distinguish between several situations. If the right-hand side of the rule is exhausted, there are two possibilities: the input segment is also exhausted, in which case the left-hand side of the dotted goal has been fully recognized; or it is not, in which case the dotted goal has failed and nothing needs to be done. If the left-hand side in this dotted goal has been fully recognized, we first signal this fact to **TD.knownRuleGoals** to be remembered for future use. Next we temporarily pop it off the dotted-goal stack. Since the fields **rhsUsed** and **inputUsed** are

```
public class DottedGoal extends RuleGoal {
    private int rhsUsed, inputUsed; // positions of dot in rhs and input
    public void doWork() {
        TD.dottedGoalStack.push(this);
        TD.ruleStack.push(rule);
        doAsTopOfStack();
        TD.ruleStack.pop();
        TD.dottedGoalStack.pop();
    }
    public void doWorkAfterMatch(int matchedLength) {
        // advance the dotted goal over matched non-terminal and input
        rhsUsed += 1; inputUsed += matchedLength;
        doAsTopOfStack();
        // retract the dotted goal
        rhsUsed -= 1; inputUsed -= matchedLength;
    }
    private void doAsTopOfStack() { // 'this' is top of parsing stack
        int activePos = pos + inputUsed;
        int leftoverLength = length - inputUsed;
        if (rule.rhs.length == rhsUsed) { // rule exhausted
            if (leftoverLength == 0) {    // input exhausted
                TD.knownRuleGoals.recordParsing(this);
                TD.dottedGoalStack.pop();
                ((RuleGoal)this).doWorkAfterDone();
                TD.dottedGoalStack.push(this);
            }
        } else {
            Symbol rhsAtDot = rule.rhs[rhsUsed];
            if (leftoverLength > 0) {
                Symbol inputAtDot = TD.input.symbolAt(activePos);
                if (rhsAtDot.equals(inputAtDot)) doWorkAfterMatch(1);
            }
            for (int len = 0; len <= leftoverLength; len++)
                (new Goal(rhsAtDot, activePos, len)).doWork();
        }
    }
}
```

Fig. 17.8. The class **DottedGoal**

now meaningless, the dotted goal reverts to being a rule goal, and a recognized one at that, and we continue our search by applying **doWorkAfterDone()** to it. (The qualifier **((RuleGoal)this)** in front of **doWorkAfterDone()** is actually superfluous, but serves to emphasize that we are now back dealing with a rule goal.) When done, we restore the old situation by pushing the present dotted goal back on the stack.

This method `RuleGoal.doWorkAfterDone()` (page 557) checks if the stack is empty. If it is, the start symbol, which was the `lhs` of the goal that was the first to be stacked, has been matched to the entire input, so a parsing has been found. If it is not, `doWorkAfterDone()` obtains the dotted goal on top of the stack, and calls its `doWorkAfterMatch(int matchedLength)` method. This method bumps the top of the stack over the matched length, and continues the search by calling `doAsTopOfStack()`, as before.

In this way both the flow of control and our explanation return to `DottedGoal`, where we were discussing `doAsTopOfStack()`. If we enter the `else` branch of the if-statement, the right-hand side of the rule is not exhausted, and there is a symbol `rhsDot` after the dot in it. Now there are several possibilities. The symbol `rhsDot` may be a terminal, in which case we want to check it against the input symbol. We first check if there is more input, and then compare it to the input symbol at `activePos`. If they match we call `doWorkAfterMatch(1)` to bump the top of the stack over one token, and continue searching; if they do not the goal has failed. If `rhsDot` is a non-terminal, it is tried against increasingly longer chunks of what remains of the input segment, by creating a new `Goal` for each chunk. This brings us back to the level of `Goal`s, and a new cycle in the top-down search can start.

Since the presented parser does not distinguish between terminals and non-terminals, we have a problem here: we cannot know if we should try the `if (rhsAtDot···` statement (for a terminal) or the `for (int len = 0; len <=···` statement (for a non-terminal). But since they are part of a search process we can solve this by just trying them both. One consequence of this is that we cannot prevent new `Goal`s from being created for a terminal symbol; since there is no syntax rule for a terminal, the for-loop in `Goal.doWork()` will find no match. Another consequence is that the program can also handle input in which some non-terminal productions have already been recognized.

We see that the methods `doWork()`, `doWorkAfterMatch(int)`, and the if-branch in `doAsTopOfStack()` are similar in structure: they all start with a modification of the situation, perform a recursive search, and then carefully restore the original situation when the recursive search returns. This technique allows all parsings to be found.

Note that besides the parse stack and the rule stack, there is also a search stack. Whereas the first two are explicit, the third is implicit and is contained in the Java recursion stack.

17.3.3 Handling Left Recursion

As explained in Section 6.3.1, a top-down parser will loop on a left-recursive grammar and the problem can be avoided by making sure that no new goal is accepted when that same goal is already being pursued. This is achieved by the test `TD.dottedGoalStack.contains(this)` in `RuleGoal.doWork()`. When a new goal is about to be put on the parse stack, `RuleGoal.doWork()`

tests if the **DottedGoalStack** already contains the goal **this**. If it does, the goal is not tried for the second time, and **RuleGoal.doWork()** returns immediately.

The above program was optimized for brevity and, hopefully, for clarity. With an empty implementation of the methods in the class **KnownRuleGoals**, however, it may require an amount of time that is exponential in the length of the input. The optimization in the following section remedies that.

17.3.4 Parsing in Polynomial Time

An effective and relatively simple way to avoid exponential time requirement in a context-free parser is to equip it with a *well-formed substring table*, often abbreviated to *WFST*. A WFST is a table which holds all partial parse trees for each substring (segment) of the input string; it is very similar to the table generated by the CYK algorithm. It is can be shown that the amount of work needed to construct the table cannot exceed $O(n^{k+1})$ where n is the length of the input string and k is the maximum number of non-terminals in any right-hand side. This takes the exponential sting out of the depth-first search.

The WFST can be constructed in advance (which is what the CYK algorithm does), or while parsing proceeds ("on the fly"). We shall do the latter here. Also, rather than using a WFST as defined above, we shall use a *known-parsing table*, which holds the partial parse trees for each **RuleGoal**, i.e., each combination of a grammar rule and a substring of the input. These two design decisions have to do with the order in which the relevant information becomes available in the parser described above.

An implementation of the known-parsing table is shown in Figure 17.9. The class **KnownRuleGoals** supplies the methods **startNewParsing(RuleGoal)**, **recordParsing(RuleGoal)** and **doWork(RuleGoal)**.

The parser interacts with the known-parsing table **TD.knownRuleGoals** only in a few places. The first place is in **RuleGoal.doWork()**, where a call is made to **knownRuleGoalTable.containsKey(this)**. This method accesses the known-parsing table to find out if the rule goal **this** has been pursued before. When called for the very first time, it will yield **false** since there are no known parsings yet. So we skip the statements controlled by the if, and land at the call of **startNewParsing(this)**. This prepares the table for the recording of the zero or more parsings that will be found for **this**. Once the parsings have been found and the test is made a second time, it succeeds, and, rather than trying the rule goal again, a call to **TD.knownRuleGoals.doWork(this)** is made, which produces the zero or more parsings from the known-parsing table. The last interaction between the parser and the known-parsing table is in **DottedGoal.doAsTopOfStack()**, where a call of **recordParsing()** is used to enter the discovered parsing.

The rule goals are recorded in a three-level data structure. The first level is the hash table **knownRuleGoalTable**, indexed by **RuleGoal**s; its elements are objects of class **KnownRuleGoal**. A **KnownRuleGoal** has **ruleGoal** field and a **knownParsingSet** field, which is a vector of objects of class **KnownParsing** and which forms the second level. Each **KnownParsingSet** contains a parse tree

```
import java.util.ArrayList;
import java.util.Hashtable;
public class KnownRuleGoals {
  Hashtable<RuleGoal,KnownRuleGoal> knownRuleGoalTable
      = new Hashtable<RuleGoal,KnownRuleGoal>();
  Hashtable<RuleGoal, Integer> startParsingTable
      = new Hashtable<RuleGoal, Integer>();
  public void startNewParsing(RuleGoal ruleGoal) {
    startParsingTable.put(ruleGoal, new Integer(TD.ruleStack.size()));
    knownRuleGoalTable.put(ruleGoal, new KnownRuleGoal(ruleGoal));
  }
  public void recordParsing(RuleGoal ruleGoal) {
    knownRuleGoalTable.get(ruleGoal).record();
  }
  public void doWork(RuleGoal ruleGoal) {
    knownRuleGoalTable.get(ruleGoal).doWork();
  }
  private class KnownRuleGoal {
    RuleGoal ruleGoal;
    ArrayList<KnownParsing> knownParsingSet =
        new ArrayList<KnownParsing>();
    void record() {
      knownParsingSet.add(new KnownParsing());
    }
    void doWork() {
      for (int i = 0; i < knownParsingSet.size(); i++) {
        knownParsingSet.get(i).doWork();
      }
    }
    private class KnownParsing {
      Rule [] knownParsing;
      KnownParsing() {
        int stackSizeAtStart =
          startParsingTable.get(ruleGoal).intValue();
        int stackSize = TD.ruleStack.size();
        knownParsing = new Rule[stackSize - stackSizeAtStart];
        for (int i = stackSizeAtStart, j = 0; i < stackSize; i++, j++) {
          knownParsing[j] = TD.ruleStack.elementAt(i);
        }
      }
      void doWork() {
        int oldStackSize = TD.ruleStack.size();
        for (int i = 0; i < knownParsing.length; i++) {
          TD.ruleStack.push(knownParsing[i]);
        }
        ruleGoal.doWorkAfterDone();
        TD.ruleStack.setSize(oldStackSize); // pop all
      }
    }
  }
}
```

Fig. 17.9. Implementation of the known-parsing table

for the described goal, represented as an array of **Rule**s called **knownParsing**; this is the third level.

There is also a hash table **startParsingTable**, indexed with **RuleGoal**s, which holds the level of the rule stack at the time when work on the **RuleGoal** was started.

With this knowledge of the data structure we can easily understand the methods in **KnownRuleGoals**. The method **startNewParsing(RuleGoal)** records the size of the rule stack in the hash table **startParsingTable**, with the rule goal as an index. The idea is that when a complete parsing is found for the rule goal, the segment of the rule stack between the recorded size and the actual size contains that parsing. It also puts an empty **KnownRuleGoal** in the known-parsing table; if no parsings are found for the rule goal, this empty entry serves to record that fact.

When the main mechanism of the parser has found a parsing for the rule goal (in **DottedGoal.doAsTopOfStack()**) it calls **recordParsing()**. This method is very simple: it retrieves the **KnownRuleGoal** for the rule goal, and applies its **record()** to it.

The method **record()** in **KnownRuleGoal** creates a new **KnownParsing** and adds it to the **knownParsingSet**. The constructor for the **KnownParsing** retrieves from the hash table **startParsingTable** the rule stack size at the time the parsing for this rule goal started and the present rule stack size, creates the array **knownParsing[]** and copies the parsing into it from the segment of the rule stack between the present stack size and **stackSizeAtStart**. The parsing is now safely stored in the array in the vector in the hash table.

Note that as long as the rule goal is under construction, it is also on the parse stack. This means that the left-recursion protection test **TD.dottedGoalStack.contains(this)** in **RuleGoal** cannot yield **false** until after the rule goal has been rebmoved from the parse stack, i.e., after all its parsings have been found. So when **TD.knownRuleGoals.doWork(this)** is called eventually, we can be sure that it can reproduce all possible parsings.

This brings us to the method **doWork(RuleGoal)** in **KnownRuleGoals**; it retrieves the **KnownRuleGoal** for the rule goal, and applies its **doWork()** to it. This method steps through the **knownParsingSet** and applies **doWork()** to each **KnownParsing** in it. The method **KnownParsing.doWork()** is the first in the known-parsing class that directly contributes to the parsing activity. It copies the parsing stored in **knownParsing** to the rule stack as if it had been performed normally, signals to the rule goal that the rule goal has been recognized, and removes the copied parsing. The net result of the call to **TD.knownRuleGoals.doWork(this)** is that **RuleGoal.doWorkAfterDone()** is called exactly as many times and in the same situations as if the rule goal were parsed without the use of the known-parsing table, only much faster.

It will be obvious that copying a ready-made solution is much more efficient than reconstructing that solution. That it makes the difference between exponential and polynomial behavior is less obvious, but true nevertheless. When applied to the input strings **())** and **(((()))))**, the parser without the known-parsing table tries 41624

rules for the `UserGrammar4` example, the parser with it only 203. A parser that does not use the known-parsing table can be obtained simply by putting `false &&` in front of the test of `containsKey` in `RuleGoal.doWork()` (page 557).

17.4 Programming Language Paradigms

A programming paradigm is a mind set for formulating and solving programming problems. A paradigm is characterized by a single principle, a finite set of concepts to support the principle, an infinite set of methods to apply the concepts to solve the problem, and, hopefully, a user-community-cum-culture, with books, user groups, etc., to spread the word. There are four major programming paradigms: imperative ("do this, then do that"); object-oriented ("everything is an object, with buttons (methods) on the outside"); functional ("everything is a function from input to output"); and logic ("everything is a set of relations held together by Horn clauses").

There is also parallel and distributed programming, but that is rather a — hopefully beneficial — restriction than a programming paradigm, in that it affects the nature of the algorithms expressible in it. It is covered in Chapter 14.

Although in principle anything programmable can be programmed in any paradigm, some combinations are much more convenient than others, and it is interesting to see how the different paradigms relate to the various programs arising in parser writing. Figures 17.1 and 17.2 show the four kinds of programs that are likely to be produced by a parser writer: the interpreting parser, the parser generator, the generated table-driven parser, and the compiled parser.

Interpreters and parser generators are just programs, no different in their nature than any other programs; they can be written in any language in any paradigm the programmer finds convenient. An example of an interpreter in Java was given in Section 17.3.

Table-driven parsers do not contain much in the way of programming: just a simple loop accessing a table. The imperative paradigm is no doubt the best for this; there are no obvious objects, and functional and logic languages are not very good at handling large matrices. On the other hand, table-driven parsers often contain semantic routines, and these may dictate the programming language and the paradigm.

We now turn to compiled parsers, parsers for which code in some programming language must be generated. One powerful method for creating parser code is to generate a parsing routine for each non-terminal, as in recursive descent (Sections 6.6 and 8.2.6). The idea was first suggested in the beginning of 1961 by Grau [332], but he could not implement the idea because he had no compiler capable of handling recursive routines. The first explicit description is by Lucas [41] later that year. The idea was formalized by Knuth [43] in 1971. In the next few sections we discuss the generation of recursive descent parsers in the four major paradigms.

17.4.1 Imperative and Object-Oriented Programming

Constructing a deterministic (LL(1)) parser by compiled recursive descent is very simple in an imperative language, once the look-ahead sets have been computed;

Section 8.2 describes the technique. Doing the same in an object-oriented language while respecting the object-oriented paradigm is much harder. The reason is that one would like to encapsule all information about a non-terminal *A* in a single object, **parse_***A*, but the problem is that the look-ahead sets of the alternatives of *A* belong to *A* but depend on many other non-terminals. One can of course program around this but the resulting code is awkward.

Two possible solutions to this problem have been suggested. The first is by Koskimies [342], who lets each **parse_***A* start with empty look-ahead sets. The first time *A* is called and has to decide between its alternatives, it calls them one by one. The routines for the alternatives then report back their look-ahead sets, and *A* assembles its from theirs. The second is by Metsker [357], who hides the details in a toolkit which defines the classes **Repetition**, **Sequence**, **Alternation** and **Word**. A parser can then be constructed in object-oriented fashion from these.

17.4.2 Functional Programming

Although functional languages are less than great for deterministic, table-driven parsers (but see Leermakers et al. [346]), they are very convenient for backtracking recursive descent parsers. The three mechanisms in a CF grammar are concatenation, choice and identification, as explained on page 24. The idea is to have the first two of these mechanisms as higher-order functions in our parser; the identification comes free of charge through the programming language.

A higher-order function is a function that takes functions as its parameters; in our parser these parameters will be functions that parse given non-terminals, and the two higher-order functions, called *combinators*, coordinate the parsing. *Combinator parsing* was first described by Frost [344] and Hutton [345], but we shall follow here Frost and Szydlowski [353], who give the following short, self-contained example.

The code for a parser for the grammar **S→aSS**|ε in the functional language Haskell using combinators is

```
s = (a `and_then` s `and_then` s) `or_also` empty
a = term 'a'
```

We see that the translation is immediate (with concatenation represented by **`and_then`** and choice by **`or_also`**), but it does require a lot of explanation.

The Haskell function for parsing a non-terminal *A* accepts one parameter as input, a list of strings. It picks up each string from the list in succession and tries to find one or more prefixes in it that correspond to *A*. If it finds any, it adds the strings that remain after the prefixes have been removed to the output list; otherwise it adds nothing to the list.

Suppose *A* produces **[a|b]*b** and we call its parsing function with a list of three strings **["aab","aa","abbaba"]**, then the output is a list of four strings: **["","a","aba","baba"]**. The first, empty string results from **"aab"**, from which the full **aab** has been removed as a prefix; the second input string has no prefix that matches *A*, so it does not feature in the output; and the last three strings result

from **"abbaba"**, from which **abbab**, **abb**, and **ab** have been removed as prefixes successively. (Actually the Haskell system produces the strings in a different order, due to its particular search order.)

There are two things to be noted here. The first is that a completed parsing results in an empty string to be appended to the list of strings, and vice versa an empty string indicated a successful parsing; this is the way we shall interpret the final result. The second is that the output list of strings may be shorter or longer or equally long as the input list, depending on how many parsings fail, deleting strings, and how many are locally ambiguous, producing more than one string. The strings themselves can only get shorter, or keep the same length. So, although the routine for *A* seems to act as a filter, letting only those strings pass whose prefixes match *A*, it differs from a filter in that it can duplicate the things passing through it.

Now we must implement the combinators **`and_then`** and **`or_also`**; the backquotes around the names indicate to the Haskell system that they are infix functions. The other two functions, **term** and **empty**, will be defined later. The **`or_also`** combinator is simple:

```
(p `or_also` q) inputs = (p inputs) ++ (q inputs)
```

which says that the function **(p `or_also` q)** is applied to the parameter **inputs** by applying the function **p** to it, then applying **q** to it, and concatenating (**++**) the two resulting lists. Note that the input list gets copied here: both **p** and **q** start with the same lists of inputs and their contributions are combined.

The combinator **`and_then`** is more complicated (see Problem 17.5):

```
(p `and_then` q) inputs | (r == []) = []
                        | (r /= []) = (q r)
                          where
                              r = (p inputs)
```

It features a local variable **r**, which is computed first, by applying **p** to the list of strings. So **r** is the list of strings that remain after a prefix matching **p** has been removed from them. If that list is empty (that is, no string had a prefix matching **p**), we return the empty list; if it is not, we apply **q** to it and return the result. Any string in that result corresponds to a string in **inputs** that had prefixes matching **p** followed by **q** removed from it.

Remarkably, the function that parses a terminal symbol (that is, removes it as a prefix) looks even more forbidding:

```
term c []     = []
term c (s:ts) = (term_c s) ++ (term c ts)
                where
                    term_c ""                = []
                    term_c (c1:s) | (c1 == c) = [s]
                                  | (c1 /= c) = []
```

but that is because it has to take the list apart and then the strings in it, to get at the first tokens of these strings, and then reassemble the lot. The first definition of **term c inputs** says that if the input is the empty list, so is the output. The second

says that if the input can be split into a string **s** and the rest of the strings **ts**, then the output is composed by applying an auxiliary function **term_c** to the string **s** and concatenating the result with the result of the original function working on the rest of the strings, **ts**.

The definition of the function **term_c** follows in the **where** section. The first line says that if its parameter is the empty string, the result is the empty list. The next two lines test for the first token of the parameter: if it is **c**, the token we want to match, we return the rest of the string packed in a list, otherwise we return the empty list.

Note how both the empty string and the non-matching string are turned into the empty list. The empty list is then concatenated with the rest of the list, which makes it disappear from the game: the empty list indicates failure. On the other hand, a call of **term c s** where **s** is **"c"** results in a list of one element, the empty string, **[""]**. This empty string is just a new element of the output list; it stays in the game, and can in the end signal success. "Emptiness" is a subtle thing...

The function representing ε is very simple:

```
empty inputs = inputs
```

It just copies the input list.

We now have a complete program, and can call the parsing routine for **S** on an input string, say **aaa**. To do this, we write

```
s ["aaa"]
```

and the system responds with

```
["","","","a","","","a","aa","aaa"]
```

This reports five successful parsings and four failures. This technique can handle any non-left-recursive CF grammar.

Some of the above functions can be written much more compactly and elegantly in Haskell, but we have chosen the forms shown here because they require the least knowledge of Haskell.

The above describes the bare bones of functional parsing, and many additions and improvements need to be made. We will mention a few here, but recent literature supplies many more. A filter is needed to clean out failed attempts. A semantics mechanism must be added, since the "parser" we constructed above is actually a recognizer; Hutton [345] describes how to do that. The naive parser has exponential time complexity, which can be reduced to cubic by memoization; see for example Frost [350] or Johnson [351]. Even more efficiency can be gained by doing partial evaluation, as explained by Sperber and Thiemann [356].

Ljunglöf [358] describes extensively how to program parsers in Haskell, and includes detailed code for Earley and chart parsers. Several books on functional programming have a section on parser writing, for example Thompson [413, Sect. 17.5].

The functional paradigm has also made great contributions to the field of natural language parsing, in the form of the many natural language parsers written in Lisp.

17.4.3 Logic Programming

If functional languages are a good vehicle for compiled parsers, logic languages, more in particular Prolog, are an even better fit. The examples in this book (Definite Clause Grammars in Section 6.7, Cancellation Parsing in Section 6.8, and parsing VW grammars in Section 15.2.3), and the many papers listed in the sections on non-Chomsky systems ((Web)Section 18.2.6), natural language parsing (18.3.5), parser writing (18.3.1), and parsing as deduction (18.3.4), are ample proof of that. Prolog also plays a considerable role in natural language parsing.

Prolog has two major advantages over most other paradigms: the search mechanism is built-in, handling the context-free part; and the semantics can be manipulated conveniently with logic variables, handling the context-sensitive part. Logic variables are subject to unification, a very powerful data-manipulation mechanism not easily programmed in the other paradigms.

Only the constraint programming paradigm is more powerful, but it is experimental, and no complete implementation technique for it is known or perhaps even possible. Still, progress is being made on it, and it finds its way into parsing; see, for example, Morawietz [373] or Erk and Kruijff [374].

17.5 Alternative Uses of Parsing

Parsing is the structuring of text according to a grammar, no more, no less. In that sense there cannot be alternative uses of parsing. Still, some applications of parsing are unusual and perhaps surprising. We will cover three of them here: data compression, machine instruction generation in compilers, and support of logic languages. In the first two, text structuring is still prominent, but the third may qualify as "alternative use of parsing techniques".

17.5.1 Data Compression

Files can be compressed only if they contain redundancy, but most files people use have some. Usually this redundancy is internal: if we find that a file contains the word **`aardvark`** many times, we can replace it by **`@23`** provided **`@23`** does not occur otherwise. This is the kind of redundancy that is exploited by various *zip* programs. But redundancy can also be external: if both the sender and the receiver know that a file contains a tax declaration, only the numbers with the names of the boxes they go into have to be stored in the file, not all the surrounding text. This kind of redundancy is exploited by data bases. In both cases the important point is that we know something about the file, either by inspection or as advance knowledge.

Knowing that a file conforms to a CF grammar amounts to a lot of information, and over the last decade progress has been made to utilize that knowledge. Java applets that have to be sent over the Internet have been especially interesting targets for the technique. Such programs are transmitted in source or byte code, so they can

be subjected to user security checks before they are run on the receiver's computer (Evans [366]).

The idea is to store the sequence of rule numbers that produced the file — its leftmost derivation, see Section 3.1.3 — rather than the file itself, in what is called *grammar-based compression*. In terms of Java this means that, given that rule 75 of the grammar is

```
ForStatement  →  for ( ForInitOption ; ExpressionOption ;
                 ForUpdateOption ) Statement()
```

it is cheaper to store **75,1** for non-terminal 75, alternative 1, than the produced text **for (... ; ... ; ...)**. The **75,1** can be stored in 11 bits: 7 for the non-terminal number since there are fewer than 128 non-terminals in the Java grammar, and 4 bits for the alternative number since there are at most 16 alternatives to any non-terminal. The produced text form would cost 7 bytes, = 56 bits, so in this case storing the leftmost derivation saves 45 bits, which is 80.4%.

Actually it is better than that: leftmost derivation implies leftmost production at the receivers end, and in leftmost production we know at any moment which non-terminal we are going to expand: the leftmost one in the sentential form. So we do not have to store the non-terminal number at all, and 4 bits for the alternative number suffice, raising our savings to 52 bits, 92.9%. Even better, since the non-terminal **ForStatement** has only one alternative, we do not need to store the alternative number at all, bringing us an untoppable savings of 100%! Of course, in a sense this is cheating since the information that the next non-terminal is **ForStatement** was supplied by the preceding **Statement**, at a cost of 4 bits since it has 16 alternatives. Still, savings can be considerable, as we will see.

This juggling of alternative numbers, bits, and compression rates soon gets messy and a more systematic approach is needed. Also, we want to compare the performance of grammar-based compression to the standard Lempel–Ziv compression.

As a highly simplified example we will use files conforming to the grammar

```
1.  S_S  →  a S a
2.       |  b S b
3.       |  c
```

We start with a random file obeying the above grammar, 1 000 001 bytes long, and starting with **bababba...**. Since the file contains only three different tokens, and one of them only once, we expect the file to compress well under traditional techniques and indeed *gzip* reduces its size to 159 107 bytes (84.1%) (see the table in Figure 17.10).

The compressed version is now constructed as follows. LL(1) parsing of the file immediately reveals that the top-most rule of the parse tree is **S→bSb**, which is rule number 2. There are three rules to **S**, so specifying the alternative will require 2 bits. Since there is no rule 0, the three rules can be specified by the 2-bit integers 0, 1, and 2. So for the first byte of the file, the two bits 01 are output. The next byte is an **a**, which is produced by rule 1, so 00 is appended. The next two bytes yield 0100, which completes one byte of the compressed file: 01000100. This process is repeated until after 500 000 bytes we reach the **c**; in the meantime we have output 500 000/4

Object	Size in bytes	Compression rate
Demo file	1 000 001	0%
Demo file, zipped	159 107	84.1%
With naive parsing	125 001	87.5%
With naive parsing, zipped	71 909	92.8%
Parsed with 1 expanded rule	93 793	90.6%
Parsed with 1 expanded rule, zipped	67 496	93.3%
Parsed with 253 expanded rules	62 684	93.7%
Parsed with 253 expanded rules, zipped	62 717	93.7%

Fig. 17.10. The effects of various stages of grammar-based compression

= 125 000 bytes in the compressed file. The `c` is parsed by rule 3, so we output the bits 10. Now the second half of the input file has already been completely predicted by the LL(1) parser, so no more rule numbers are produced. We fill up the last byte with arbitrary bits, and we are done. The resulting size is 125 001 bytes (87.5%), which is already better than *gzip* did.

The receiving program starts with `S` as the initial form of the reconstructed file. It then reads the first byte of the compressed file, 01000100, extracts the first two bits, concludes that it needs to apply rule number 2 of non-terminal `S`, and replaces the reconstructed form by `bSb`. The next 2 bits turn it into `baSab`, etc. When the code 10 is found, it identifies rule number 3, and the `S` gets replaced by `c`. Now there is no non-terminal left in the reconstructed form, so the process stops, the last few bits in the input are ignored and the reconstructed form is written to file.

Since Lempel–Ziv compression is completely different from grammar-based compression, it is tempting to apply it to the compressed file. Indeed the size reduces further, to 71 909 bytes, and it is easy to see why. The two-bit integers we were writing to the compressed file can only have the values 00, 01, and 10, and the last value occurs only once; so we are using only 50+ε% of the capacity. This suggests that we would have done better with a rule with 4 alternatives rather than 3. That can be arranged, by substituting one non-terminal A in the right-hand side of some rule B by a right-hand side of A. We can, for example substitute `S→aSa` into `S→bSb`, resulting in `S→baSab`; this will be our rule number 4. The results are shown in the third section of Figure 17.10, and we see that it helps considerably; but additional zipping still works, so there is still redundancy left. (Another way to remedy the bad fit is by using adaptive arithmetic coding, as reported by Evans [366].)

If four alternatives are better than three, more might be even better; and it is. Evans [367] shows that it is efficient to substitute out non-terminals until they all have 256 alternatives. Then each alternative number fits in exactly one byte, which speeds up processing in both the compressing and the decompressing side, and no space is lost. This immediately raises the question which non-terminals to substitute in which

right-hand sides. Evans gives heuristics, possibly involving analyzing the input file in advance, but since we want to keep it simple in our example, we substitute the rules **S→aSa** and **S→bSb** in each other until we have rules that start with all combinations of 8 **a**s and **b**s. That yields 256 rules, and because we still need our first three rules, we just discard the last three rules. This means that our grammar now looks as follows:

```
1.    Ss  →  a S a
2.           |  b S b
3.           |  c
4.           |  a a a a a a a a S a a a a a a a a
5.           |  a a a a a a a b S b a a a a a a a
                ...
255.         |  b b b b b a b b S b b a b b b b b
256.         |  b b b b b b a a S a a b b b b b b
```

When the input starts with one of the discarded combinations, for example **bbbbbbab**, or when the **c** is among them, rules 1 or 2 take over. Using this grammar reduces the size to 62 684 bytes (93.7%), which is very close to the theoretical minimum of 62 501 (1 bit for each **a** or **b** in the first half, + 1 bit for the **c**). It is gratifying to see that we have finally reached a compression that cannot be improved by an additional application of *gzip*.

In the above explanation we have swept an important problem under the rug: after the substitutions the grammar is no longer LL(1); it is even ambiguous. There are several ways to solve this. We observe that the above grammar is LL(8), provided dynamic conflict resolvers are attached to rules 1 and 2 to avoid these rules when possible. It is not inconceivable that such an adaptation can be automated; see Problem 17.6. Evans [367] shows how to rig an Earley parser so it always recognizes the longest possible sequence. And even in the absence of such solutions, spending considerable time compressing a file is worth while, when the result is used sufficiently often.

Two notes: Many papers on data compression use the term "parsing" in the sense of repeated string recognition, and as such these techniques do not qualify as "applications of parsing". And for readers who read Russian, some papers on grammar-based data compression are in Russian, for example Kurapova and Ryabko [352].

17.5.2 Machine Code Generation

A large part of program code in imperative languages consists of arithmetic expressions. The compiler analyses these expressions and makes all implicit actions in them explicit; examples are indirection, subscripting, and field selection. These explicit expressions, which are part of the intermediate code (IC) in the compiler, very quickly get more complicated than one would expect. For example, the integer expression **a[b]** is converted to something like **M[a+4×b]**: the value of **b** must be multiplied by 4 since integers are 4 bytes long, the address of the array **a** must be added to it, and the memory location at that address must be read.

In the end such intermediate code expressions must be converted to machine instructions, which can also be seen as expressions. For example, most machines have an *add constant* instruction, which adds a constant to a machine register. It could be written **ADDC** R_n**,c** and be represented by the expression R_n**:=c+**R_n.

One way to translate from intermediate code to machine code is to generate the IC expressions according to an IC grammar and to reparse this stream according to a machine code grammar. That is exactly what Glanville and Graham [336] do; they use a modified SLR(1) parser for the process. The technique is variously referred to as Glanville–Graham and Graham–Glanville, which again goes to show that techniques should not be named after people. We will call it *expression-rewriting code generation*, in line with the better known tree-rewriting code generation.

Intermediate and machine code grammars are large and repetitive: the SLR(1) parser for the very simple example in Glanville and Graham's paper already has 42 states. We shall therefore give here a totally unrealistic example and just sketch the process; for more details we refer to the above paper and to literature on the Internet.

Suppose we have a machine with the following six machine instructions:

Name	Rule	Assembler	Cost
Add constant **c**	**R[n] → + c R[n]**	**ADDC** R_n**,c**	1
Multiply by $-128 \leq$ **c** ≤ 127	**R[n] → × c R[n]**	**MULSC** R_n**,c**	2
Multiply by constant **c**	**R[n] → × c R[n]**	**MULC** R_n**,c**	3
Load address of variable **v**	**R[n] →** A_v	**LA** R_n**,v**	1
Load address of an element of array **a** (**c**$_2$=4)	**R[n] → +** A_a **× c R[i]**	**LAAE** R_n**,a,[**R_i**]**	3
Load value from memory	**R[n] → @ R[i]**	**LD** R_n**,M[**R_i**]**	3

The expressions are presented in prefix form: the R_n**:=c+**R_n above shows up as **R[n] →+cR[n]**. There are two instructions for multiplication, one with a small (one-byte) constant, and another with any constant. The **LAAE** instruction Loads the Address of an Array Element. The **LD** instruction loads one register with the value of the memory location at (**@**) another register. The column marked "Assembler" shows the machine instructions to be generated for each rule. "Cost" specifies the cost in arbitrary units. The grammar contains two kinds of context or semantic conditions. The first is that the register numbers must be substituted consistently: the first line is actually an abbreviation for

Add constant	**R[1] → + c R[1]**	**ADDC** R_1**,c**	1
Add constant	**R[2] → + c R[2]**	**ADDC** R_2**,c**	1
...			

The second is the condition $-128 \leq$ **c** ≤ 127 on the **MULSC** instruction and the **c**$_2$=4 in **LAAE**.

The grammar is ambiguous and certainly not SLR(1). There are many ways to resolve the conflicts; we will resolve shift-reduce conflicts by shifting and re-

duce/reduce conflicts by reducing with the longest reduce with the lowest cost which is compatible with the semantic restrictions.

We now return to our source code expression **a[b]**, which translates into the intermediate code expression **@+A**$_a$×**4@A**$_b$. Here **A**$_a$ is a constant, equal to the machine address of the first word of the array **a**, and **A**$_b$ is the address of the variable **b** in memory. When this expression is parsed, the parser goes through a number of shift/reduce conflicts, and shifts it completely onto the stack.

The further actions are shown in Figure 17.11. Initially only one reduction can

Stack	Action	Machine instruction	Cost
@ + A$_a$ × **4 @ A**$_b$	reduce, Load addr. of var.	**LA R**$_1$**,b**	**1**
@ + A$_a$ × **4 @ R**$_1$	reduce, Load value from mem.	**LD R**$_2$**,M[R**$_1$**]**	**3**
@ + A$_a$ × **4 R**$_2$	reduce, Load addr. of ar. elem.	**LAAE R**$_3$**,a[R**$_2$**]**	**3**
@ R$_3$	reduce, Load value from mem.	**LD R**$_4$**,M[R**$_3$**]**	**3**
R$_4$	result left in **R**$_4$		**10**

Fig. 17.11. Parser actions during expression-rewriting code generation

be done, using the rule for loading the address of a variable. It replaces the **A**$_b$ on the stack by **R**$_1$, while at the same time issuing the instruction **LA R**$_1$**,b**. We see that reducing the top segment T of the stack to a register R corresponds to code which at run time puts the value of the expression corresponding to T into R: the combined action leaves the semantics unaltered — the basic tenet of code generation.

The second reduction is also forced: load value (that of **b**) from memory; to simplify matters we use a very simple register allocation scheme here: just assign a new register every time. The next stack configuration, however, has a triple reduce/reduce conflict: it can be reduced with the instructions **MULSC**, **MULC**, and **LAAE**. The first matches because the constant (4) is small, the second has no restrictions and the third matches because the constant is in range. The criterion to take the cheapest of the longest reductions that fit the restrictions leads us to use **LAAE**. The last step is again forced, and leaves the result of the expression **a[b]** in register 4, at a total cost of 10 units.

Now suppose the source expression had been **&a+5**×**b**, where **&a** is the address of **a** in a C-like notation. This would have resulted in an intermediate code expression **+A**$_a$×**5@A**$_b$. The first two steps in the reduction sequence remain the same, but in the resulting stack configuration **+ A**$_a$ × **5 R**$_2$ the constant is not 4, and **LAAE** no longer fulfills the restrictions. The other two reductions do, however, and the cheapest is chosen:

+ A$_a$ × **5 R**$_2$	reduce, Multiply by small const.	**MULSC R**$_3$**,5**	**2**
+ A$_a$ **R**$_3$	reduce, Add constant	**ADDC R**$_3$**,a**	**1**
R$_3$	result left in **R**$_3$		**9**

We see that the algorithm automatically adapts to a small change in the intermediate code expression by generating quite different code. Combined with the possibility to impose context restrictions and assign costs, expression-rewriting code generation

is a versatile tool for generating good code for expressions. (More recently it has been superseded by BURS-techniques, which rewrite trees rather than expressions.)

There are several problems with the above approach; they are all concerned with the fact that we are using an ambiguous grammar to do parsing. So we run the risk of making a decision that will turn out to block progress further on. See Glanville and Graham's paper [336] for solutions.

17.5.3 Support of Logic Languages

In Sections 6.7 and 17.4.3 we have seen how a logic language, in this case DCG-extended Prolog, can be used to implement parsing. This works both ways: it is also possible to use parsing to support the inference process involved in logic languages. Normally logic languages use a built-in top-down depth-first search, as do many parsing algorithms, but many of the latter also incorporate some breadth-first and/or bottom-up component. The idea is to use the well-balanced search techniques from the parsing scene to guide the inference process in the logic language. This usually requires imposing some restrictions on the logic languages.

Rosenblueth [370] uses chart parsers as inference systems for logic programs in which the arguments of the logic predicates can be classified as either input or output, a restriction reminiscent of that on the attributes in attribute grammars. In [371] Rosenblueth and Peralta do the same using SLR parsing. Vilain [369] exploits tabular Earley parsing to implement deduction recognition in a frame language.

17.6 Conclusion

Broadly speaking, general CF parsing is best done with a GLR(0) parser, and the same goes for general CF substring parsing. For linear-time parsing strong-LL(1) and LALR(1) are still good choices. For linear-time substring parsing Bates and Lavie's technique [214] is the prime candidate.

Parsers can perform the semantics of their input immediately, or can create parse tree(s) to be processed further. In a parser, the grammar can be interpreted, compiled into a table or compiled into program code, in that order of efficiency.

Recursive-descent parsers can be implemented efficiently in all four paradigms, imperative, object-oriented, functional, and logic, with especially the latter having remarkable properties.

Parsing can be used outside the traditional setting of matching a string to a grammar; examples are data compression, machine code generation and the support of logic languages.

Problems

Problem 17.1: How does the parser from Section 17.3 handle infinite ambiguity?

Problem 17.2: Modify the parser from Section 17.3 so it delivers a parse forest rather than a sequence of parsings. This would allow infinite ambiguity to be represented better.

Problem 17.3: The explicit dotted-goal stack in the parser from Section 17.3 can be avoided by linking each dotted goal to its parent. Modify the code in that sense.

Problem 17.4: *History:* Determine the parsing technique used by Grau in his 1961 paper [332]. That is, design an algorithm that produces Grau's table or something close, making reasonable assumptions about the grammar used.

Problem 17.5: Why can we not just define `(p 'and_then' q) inputs` as `q (p inputs)`, which is exactly what `'and_then'` seems to mean?

Problem 17.6: *Project*: Given an LL or LR grammar, redesign the corresponding linear-time parser so it is still linear-time when rules are substituted, as described in Section 17.5.1.

Problem 17.7: *Project*: Expression-rewriting code generation uses ambiguous grammars and bottom-up parsing, and requires that no grammar-conforming input be rejected, regardless of how reduce/reduce conflicts are resolved. Glanville and Graham [336] give conditions on the grammar to achieve this, but these conditions, although adequate for their purpose, seem overly restrictive. Try to find more lenient conditions and still accept any correct input, under two regimes: 1. shift/reduce conflicts are always resolved by shifting; 2. shift/reduce conflicts can be resolved any way the code generator sees fit.

18

Annotated Bibliography

The purpose of this annotated bibliography is to supply the reader with more material and with more detail than was possible in the preceding chapters, rather than to just list the works referenced in the text. The annotations cover a considerable number of subjects that have not been treated in the rest of the book.

The printed version of this book includes only those literature references and their summaries that are actually referred to in it. The full literature list with summaries as far as available can be found on the web site of this book; it includes its own authors index and subject index.

This annotated bibliography differs in several respects from the habitual literature list.

- The annotated bibliography consists of four sections:
  - Main parsing material — papers about the main parsing techniques.
  - Further parsing material — papers about extensions of and refinements to the main parsing techniques, non-Chomsky systems, error recovery, etc.
  - Parser writing and application — both in computer science and in natural languages.
  - Support material — books and papers useful to the study of parsers.
- The entries in each section have been grouped into more detailed categories; for example, the main section contains categories for general CF parsing, LR parsing, precedence parsing, etc. For details see the Table of Contents at the beginning of this book.

  Most publications in parsing can easily be assigned a single category. Some that span two categories have been placed in one, with a reference in the other.
- The majority of the entries are annotated. This annotation is not a copy of the abstract provided with the paper (which generally says something about the results obtained) but is rather the result of an attempt to summarize the technical content in terms of what has been explained elsewhere in this book.
- The entries are ordered chronologically rather than alphabetically. This arrangement has the advantage that it is much more meaningful than a single alphabetic list, ordered on author names. Each section can be read as the history of research

on that particular aspect of parsing, related material is found closely together and recent material is easily separated from older publications. A disadvantage is that it is now difficult to locate entries by author; to remedy this, an author index (starting on page 651) has been supplied.

18.1 Major Parsing Subjects

18.1.1 Unrestricted PS and CS Grammars

1. **Tanaka**, Eiichi and Fu, King-Sun. Error-correcting parsers for formal languages. *IEEE Trans. Comput.*, C-27(7):605–616, July 1978. In addition to the error correction algorithms referred to in the title (for which see [301]) a version of the CYK algorithm for context-sensitive grammars is described. It requires the grammar to be in 2-form: no rule has a right-hand size longer than 2, and no rule has a left-hand size longer than its right-hand size. This limits the number of possible rule forms to 4: $A \rightarrow a$, $A \rightarrow BC$, $AB \rightarrow CB$ (right context), and $BA \rightarrow BC$ (left context). The algorithm is largely straightforward; for example, for rule $AB \rightarrow CB$, if C and B have been recognized adjacently, an A is recognized in the position of the C. Care has to be taken, however, to avoid recognizing a context for the application of a production rule when the context is not there at the right moment; a non-trivial condition is given for this, without explanation or proof.

18.1.2 General Context-Free Parsing

2. **Irons**, E. T. A syntax-directed compiler for ALGOL 60. *Commun. ACM*, 4(1):51–55, Jan. 1961. The first to describe a full parser. It is essentially a full backtracking recursive descent left-corner parser. The published program is corrected in a Letter to the Editor by B.H. Mayoh, *Commun. ACM*, 4(6):284, June 1961.

3. **Hays**, David G. Automatic language-data processing. In H. Borko, editor, *Computer Applications in the Behavioral Sciences*, pages 394–423. Prentice-Hall, 1962. Actually about machine translation of natural language. Contains descriptions of two parsing algorithms. The first is attributed to John Cocke of IBM Research, and is actually a CYK parser. All terminals have already been reduced to sets of non-terminals. The algorithm works by combining segments of the input ("phrases") corresponding to non-terminals, according to rules $X - Y = Z$ which are supplied in a list. The program iterates on the length of the phrases, and produces a list of numbered triples, consisting of a phrase and the numbers of its two direct constituents. The list is then scanned backwards to produce all parse trees. It is suggested that the parser might be modified to handle discontinuous phrases, phrases in which X and Y are not adjacent.
The second algorithm, "Dependency-Structure Determination", seems akin to chart parsing. The input sentence is scanned repeatedly and during each scan reductions appropriate at that scan are performed: first the reductions that bind tightest, for example the nouns modified by nouns (as in "computer screen"), then such entities modified by adjectives, then the articles, etc. The precise algorithm and precedence table seem to be constructed ad hoc.

4. **Kuno**, S. and Oettinger, A. G. Multiple-path syntactic analyzer. In *Information Processing 1962*, pages 306–312, Amsterdam, 1962. North-Holland. A pool of predictions is maintained during parsing. If the next input token and a prediction allows more than one new prediction, the prediction is duplicated as often as needed, and multiple new predictions result. If a prediction fails it is discarded. This is top-down breadth-first parsing.

5. **Sakai**, Itiroo. Syntax in universal translation. In *1961 International Conference on Machine Translation of Languages and Applied Language Analysis*, pages 593–608, London, 1962. Her Majesty's Stationery Office. Using a formalism that seems equivalent

to a CF grammar in Chomsky Normal Form and a parser that is essentially a CYK parser, the author describes a translation mechanism in which the source language sentence is transformed into a binary tree (by the CYK parser). Each production rule carries a mark telling if the order of the two constituents should be reversed in the target language. The target language sentence is then produced by following this new order and by replacing words. A simple Japanese-to-English example is provided.

6. **Greibach**, S. A. *Inverses of Phrase Structure Generators*. PhD thesis, Technical Report NSF-11, Harvard U., Cambridge, Mass., 1963.

7. **Greibach**, Sheila A. Formal parsing systems. *Commun. ACM*, 7(8):499–504, Aug. 1964. "A formal parsing system $G = (V, \mu, T, R)$ consists of two finite disjoint vocabularies, V and T, a many-to-many map, μ, from V onto T, and a recursive set R of strings in T called syntactic sentence classes" (verbatim). This is intended to solve an additional problem in parsing, which occurs often in natural languages: a symbol found in the input does not always uniquely identify a terminal symbol from the language (for example, *will* (verb) versus *will* (noun)). On this level, the language is given as the entire set R, but in practice it is given through a "context-free phrase structure generator", i.e. a grammar. To allow parsing, this grammar is brought into what is now known as Greibach Normal Form: each rule is of the form $Z \to aY_1 \cdots Y_m$, where a is a terminal symbol and Z and $Y_1 \cdots Ym$ are non-terminals. Now a *directed production analyser* is defined which consists of an unlimited set of pushdown stores and an input stream, the entries of which are sets of terminal symbols (in T), derived through μ from the lexical symbols (in V). For each consecutive input entry, the machine scans the stores for a top non-terminal Z for which there is a rule $Z \to aY_1 \cdots Y_m$ with a in the input set. A new store is filled with a copy of the old store and the top Z is replaced by $Y_1 \cdots Y_m$; if the resulting store is longer than the input, it is discarded. Stores will contain non-terminals only. For each store that is empty when the input is exhausted, a parsing has been found. This is in effect non-deterministic top-down parsing with a one-symbol look-ahead. This is probably the first description of a parser that will work for any CF grammar.
A large part of the paper is dedicated to undoing the damage done by converting to Greibach Normal Form.

8. **Greibach**, S. A. A new normal form theorem for context-free phrase structure grammars. *J. ACM*, 12:42–52, Jan. 1965. A CF grammar is in "Greibach Normal Form" when the right-hand sides of the rules all consist of a terminal followed by zero or more non-terminals. For such a grammar a parser can be constructed that consumes (matches) one token in each step; in fact it does a breadth-first search on stack configurations. An algorithm is given to convert any CF grammar into Greibach Normal Form. It basically develops the first non-terminal in each rule that violates the above condition, but much care has to be taken in that process.

9. **Griffiths**, T. V. and Petrick, S. R. On the relative efficiencies of context-free grammar recognizers. *Commun. ACM*, 8(5):289–300, May 1965. To achieve a unified view of the parsing techniques known at that time, the authors define a non-deterministic two-stack machine whose only type of instruction is the replacement of two given strings on the tops of both stacks by two other strings; the machine is started with the input on one stack and the start symbol on the other and it "recognizes" the input if both stacks get empty simultaneously. For each parsing technique considered, a simple mapping from the grammar to the machine instructions is given; the techniques covered are top-down (called top-down), left-corner (called bottom-up) and bottom-up (called direct-substitution). Next, look-ahead techniques are incorporated to attempt to make the machine deterministic. The authors identify left recursion as a trouble-spot. All grammars are required to be ε-free. The procedures for the three parsing methods are given in a Letter to the Editor, *Commun. ACM*, 8(10):594, Oct 1965.

10. **Younger**, Daniel H. Recognition and parsing of context-free languages in time n^3. *Inform. Control*, 10(2):189–208, Feb. 1967. A Boolean recognition matrix R is constructed in a bottom-up fashion, in which $R[i,l,p]$ indicates that the segment of the input string starting at position i with length l is a production of non-terminal p. This matrix can be filled in $O(n^3)$ actions, where n is the length of the input string. If $R[0,n,0]$ is set, the whole string is a production of non-terminal 0. Many of the bits in the matrix can never be used in any actual parsing; these can

be removed by doing a top-down scan starting from $R[0,n,0]$ and removing all bits not reached this way. If the matrix contains integer rather than Boolean elements, it is easy to fill it with the number of ways a given segment can be produced by a given non-terminal; this yields the ambiguity rate.

11. **Dömölki**, Bálint. A universal compiler system based on production rules. *BIT*, 8(4):262–275, Oct. 1968. The heart of the compiler system described here is a production system consisting of an ordered set of production rules, which are the inverses of the grammar rules; note that the notions "left-hand side" (lhs) and "right-hand side" (rhs) are reversed from their normal meanings in this abstract. The system attempts to derive the start symbol, by always applying the first applicable production rule (first in two respects: from the left in the string processed, and in the ordered set of production rules). This resolves shift/reduce conflicts in favor of reduce, and reduce/reduce conflicts by length and by the order of the production rules. When a reduction is found, the lhs of the reducing rule is offered for semantic processing and the rhs is pushed back into the input stream, to be reread. Since the length of the rhs is not restricted, the method can handle non-CF grammars.
The so-called "Syntactic Filter" uses a bitvector technique to determine if, and if so which, production rule is applicable: for every symbol i in the alphabet, there is a bitvector $B[i]$, with one bit for each of the positions in each lhs; this bit set to 1 if this position contains symbol i. There is also a bitvector U marking the first symbol of each lhs, and a bitvector V marking the last symbol of each lhs. Now, a stack of bitvectors Q_t is maintained, with $Q_0 = 0$ and $Q_t = ((Q_{t-1} >> 1) \vee U) \wedge B[i_t]$, where i_t is the t-th input symbol. Q_t contains the answer to the question whether the last j symbols received are the first j symbols of some lhs, for any lhs and j. A 1 "walks" through an lhs part of the Q vector, as this lhs is recognized. An occurrence of an lhs is found if $Q_t \wedge V \neq 0$. After doing a replacement, t is set back k places, where k is the length of the applied lhs, so a stack of Q_t-s must be maintained. If some $Q_t = 0$, we have an error. An interesting implementation of the Dömölki algorithm is given by Hext and Roberts [15].

12. **Unger**, S. H. A global parser for context-free phrase structure grammars. *Commun. ACM*, 11(4):240–247, April 1968. The Unger parser (as described in Section 4.1) is extended with a series of tests to avoid partitionings that could never lead to success. For example, a section of the input is never matched against a non-terminal if it begins with a token no production of the non-terminal could begin with. Several such tests are described and ways are given to statically derive the necessary information (FIRST sets, LAST sets, EXCLUDE sets) from the grammar. Although none of this changes the exponential character of the algorithm, the tests do result in a considerable speed-up in practice. (An essential correction to one of the flowcharts is given in *Commun. ACM*, 11(6):427, June 1968.)

13. **Kasami**, T. and Torii, K. A syntax-analysis procedure for unambiguous context-free grammars. *J. ACM*, 16(3):423–431, July 1969. A rather complicated presentation of a variant of the CYK algorithm, including the derivation of a $O(n^2 log\ n)$ time bound for unambiguous Chomsky Normal Form grammars.

14. **Earley**, J. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970. This famous paper gives an informal description of the Earley algorithm. The algorithm is compared both theoretically and experimentally with some general search techniques and with the CYK algorithm. It easily beats the general search techniques. Although the CYK algorithm has the same worst-case efficiency as Earley's, it requires $O(n^3)$ on any grammar, whereas Earley's requires $O(n^2)$ on unambiguous grammars and $O(n)$ on bounded-state grammars. The algorithm is easily modified to handle Extended CF grammars. Tomita [161] has pointed out that the parse tree representation is incorrect: it combines rules rather than non-terminals (see Section 3.7.3.1).

15. **Hext**, J. B. and Roberts, P. S. Syntax analysis by Dömölki's algorithm. *Computer J.*, 13(3):263–271, Aug. 1970. Dömölki's algorithm [11] is a bottom-up parser in which the item sets are represented as bitvectors. A backtracking version is presented which can handle any grammar. To reduce the need for backtracking a 1-character look-ahead is introduced and an algorithm for determining the actions on the look-ahead is given. Since the internal state is recomputed by vector operations for each input character, the parse table is much smaller than usual and its entries are one bit each. This, and the fact that it is all bitvector operations, makes the algorithm suitable for implementation in hardware.

16. **Kay**, M. The MIND system. In R. Rustin, editor, *Natural Language Processing*, pages 155–188. Algorithmic Press, New York, 1973. The MIND system consists of the following components: morphological analyser, syntactic processor, disambiguator, semantic processor, and output component. The information placed in the labels of the arcs of the chart is used to pass on information from one component to another.

17. **Bouckaert**, M., Pirotte, A., and Snelling, M. Efficient parsing algorithms for general context-free parsers. *Inform. Sci.*, 8(1):1–26, Jan. 1975. The authors observe that the Predictor in an Earley parser will often predict items that start with symbols that can never match the first few symbols of the present input; such items will never bear fruit and could as well be left out. To accomplish this, they extend the k-symbol reduction look-ahead Earley parser with a t-symbol prediction mechanism; this results in very general M_k^t parsing machines, the properties of which are studied, in much formal detail. Three important conclusions can be drawn. Values of k or t larger than one lose much more on processing than they will normally gain on better prediction and sharper reduction; such parsers are better only for asymptotically long input strings. The Earley parser without look-ahead (M_0^0) performs better than the parser with 1 symbol look-ahead; Earley's recommendation to use always 1 symbol look-ahead is unsound. The best parser is M_0^1; i.e. use a one symbol predictive look-ahead and no reduction look-ahead.

18. **Valiant**, Leslie G. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10:308–315, 1975. Reduces CF recognition to bit matrix multiplication in three steps, as follows. For an input string of length n, an $n \times n$ matrix is constructed, the elements of which are sets of non-terminals from a grammar G in Chomsky Normal form; the diagonal just next to the main diagonal is prefilled based on the input string. Applying transitive closure to this matrix is equivalent to the CYK algorithm, but, just like transitive closure, that is $O(n^3)$. Next, the author shows how transitive closure can be reduced by divide-and-conquer to a sequence of matrix multiplications that can together be done in a time that is not more than a constant factor larger than required for one matrix multiplication. The third step involves decomposing the matrices of sets into sets of h Boolean matrices, where h is the number of non-terminals in G. To multiply two matrices, each of their h Boolean counterparts must be multiplied with all h others, requiring $h \times h$ matrix multiplications. The fourth step, doing these matrix multiplications in time $O(n^{2.81})$ by applying Strassen's[1] algorithm, is not described in the paper.

19. **Graham**, Susan L. and Harrison, Michael A. Parsing of general context-free languages. In *Advances in Computing, Vol. 14*, pages 77–185, New York, 1976. Academic Press. The 109 page article describes three algorithms in a more or less unified manner: CYK, Earley's, and Valiant's. The main body of the paper is concerned with bounds for time and space requirements. Sharper bounds than usual are derived for special grammars, for example, for linear grammars.

20. **Sheil**, B. Observations on context-free parsing. *Statistical Methods in Linguistics*, pages 71–109, 1976. The author proves that any CF backtracking parser will have a polynomial time dependency if provided with a "well-formed substring table" (WFST), which holds the well-formed substrings recognized so far and which is consulted before each attempt to recognize a substring. The time requirements of the parser is $O(n^{c+1})$ where c is the maximum number of non-terminals in any right-hand side. A *2-form grammar* is a CF grammar such that no production rule in the grammar has more than two non-terminals on the right-hand side; nearly all practical grammars are already 2-form. 2-form grammars, of which Chomsky Normal Form grammars are a subset, can be parsed in $O(n^3)$. An algorithm for a dividing top-down parser with a WFST is supplied. Required reading for anybody who wants to write or use a general CF grammar. Many practical hints and opinions (controversial and otherwise) are given.

---

[1] Volker Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, 13:354-356, 1969. Shows how to multiply two 2×2 matrices using 7 multiplications rather than 8 and extends the principle to larger matrices.

21. **Deussen**, P. A unified approach to the generation and acception of formal languages. *Acta Inform.*, 9(4):377–390, 1978. Generation and recognition of formal languages are seen as special cases of Semi-Thue rewriting systems, which essentially rewrite a string u to a string v. By filling in the start symbol for u or v one obtains generation and recognition. To control the movements of the rewriting system, states are introduced, combined with left- or right-preference and length restrictions. This immediately leads to a 4×4 table of generators and recognizers. The rest of the paper examines and proves properties of these.

22. **Deussen**, Peter. One abstract accepting algorithm for all kinds of parsers. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 203–217. Springer-Verlag, Berlin, 1979. CF parsing is viewed as an abstract search problem, for which a high-level algorithm is given. The selection predicate involved is narrowed down to give known linear parsing methods.

23. **Graham**, S. L., Harrison, M. A., and Ruzzo, W. L. An improved context-free recognizer. *ACM Trans. Prog. Lang. Syst.*, 2(3):415–462, July 1980. The well-formed substring table of the CYK parser is filled with dotted items as in an LR parser rather than with the usual non-terminals. This allows the number of objects in each table entry to be reduced considerably. Special operators are defined to handle ε- and unit rules.
The authors do not employ any look-ahead in their parser; they claim that constructing the recognition table is pretty fast already and that probably more time will be spent in enumerating and analysing the resulting parse trees. They speed up the latter process by removing all useless entries before starting to generate parse trees. To this end, a top-down sweep through the triangle is performed, similar to the scheme to find all parse trees, which just marks all reachable entries without following up any of them twice. The non-marked entries are then removed (p. 443).
Much attention is paid to efficient implementation, using ingenious data structures.

24. **Kilbury**, James. Chart parsing and the Earley algorithm. In Ursula Klenk, editor, *Kontextfreie Syntaxen und verwandte Systeme*, volume 155 of *Linguistische Arbeiten*, pages 76–89. Max Niemeyer Verlag, Tübingen, Oct. 1984. The paper concentrates on the various forms items in parsing may assume. The items as proposed by Earley [14] and Shieber [379] derive part of their meaning from the sets they are found in. In traditional chart, Earley and LR parsing these sets are placed between the tokens of the input. The author inserts nodes between the tokens of the input instead, and then introduces a more general, position-independent item, (i, j, A, α, β), with the meaning that the sequence of categories (linguistic term for non-terminals) α spans (=generates) the tokens between nodes i and j, and that if a sequence of categories β is recognized between j and some node k, a category A has been recognized between i and k. An item with $\beta = \varepsilon$ is called "inactive" in this paper; the terms "passive" and "complete" are used elsewhere. These Kilbury items can be interpreted both as edges in chart parsing and as items in Earley parsing. The effectiveness of these items is then demonstrated by giving a very elegant formulation of the Earley algorithm.
The various versions of chart parsing and Earley parsing differ in their inference rules only. Traditional chart parsing generates far too many items, due to the absence of a top-down selection mechanism which restricts the items to those that can lead back to the start symbol. The paper shows that Earley parsing also generates too many items, since its (top-down) predictor generates many items that can never match the input. The author then proposes a new predictor, which operates bottom-up, and predicts only items that can start with the next token in the input or with a non-terminal that has just resulted from a reduction. The algorithm is restricted to ε-free grammars only, so the completer and predictor need not be repeated. Consequently, the non-terminals introduced by the predictor do not enter the predictor again, and so the predictor predicts the non-terminal of the next-higher level only. Basically it refrains from generating items that would be rejected by the next input token anyway. This reduces the number of generated items considerably (but now we are missing the top-down restriction). Again a very elegant algorithm results.

25. **Kay**, Martin. Algorithm schemata and data structures in syntactic processing. In B.J. Grosz, K. Sparck Jones, and B.L. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann, 1986. In this reprint of 1980 Xerox PARC

Technical Report CSL-80-12, the author develops a general CF text generation and parsing theory for linguistics, based (implicitly) on unfinished parse trees in which there is an "upper symbol" (top category, predicted non-terminal) α and a "lower symbol" β, the first text symbol corresponding to α; and (explicitly) on a rule selection table S in which the entry $S_{\alpha,\beta}$ contains (some) rules $A \rightarrow \gamma$ such that $A \in FIRST_{ALL}(\alpha)$ and $\beta \in FIRST(\gamma)$, i.e., the rules that can connect α to β. The table can be used in production and parsing; top-down, bottom-up and middle-out; and with or without look-ahead (called "directed" and "undirected" in this paper). By pruning rules from this table, specific parsing techniques can be selected.
To avoid duplication of effort, the parsing process is implemented using charts (Kay [16]). The actions on the chart can be performed in any order consistent with available data, and are managed in a queue called the "agenda". Breadth-first and depth-first processing orders are considered.

26. **Cohen**, Jacques and Hickey, Timothy J. Parsing and compiling using Prolog. *ACM Trans. Prog. Lang. Syst.*, 9(2):125–164, April 1987. Several methods are given to convert grammar rules into Prolog clauses. In the bottom-up method, a rule `E→E+T` corresponds to a clause `reduce([n(t),t(+),n(e)|X],[n(e)|X])` where the parameters represent the stack before and after the reduction. In the top-down method, a rule `T'→×FT'` corresponds to a clause `rule(n(t'),[t(*),n(f),n(t')]).` A recursive descent parser is obtained by representing a rule $S \rightarrow aSb$ by the clause `s(ASB) :- append(A,SB,ASB), append(S,B,SB), a(A), s(S), b(B).` which attempts to cut the input list ASB into three pieces A, S and B, which can each be recognized as an a, an s and a b, respectively. A fourth type of parser results if ranges in the input list are used as parameters: $s(X1,X4) :- link(X1,a,X2), s(X2,X3), link(X3,b,X4)$ in which $link(P,x,Q)$ describes that the input contains the token x between positions P and Q. For each of these methods, ways are given to limit non-determinism and backtracking, resulting among others in LL(1) parsers.
By supplying additional parameters to clauses, context conditions can be constructed and carried around, much as in a VW grammar (although this term is not used). It should be noted that the resulting Prolog programs are actually not parsers at all: they are just logic systems that connect input strings to parsings. Consequently they can be driven both ways: supply a string and it will produce the parsing; supply a parsing and it will produce the string; supply nothing and it will produce all strings with their parsings in the language.
See also same paper [341].

27. **Wirén**, Mats. A comparison of rule-invocation strategies in context-free chart parsing. In *Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 226–233, April 1987. Eight chart parsing predictors are discussed and their effects measured and analysed, 2 top-down predictors and 6 bottom-up (actually left-corner) ones. The general top-down predictor acts when an active edge for a non-terminal A is added at a certain node; it then adds empty active edges for all first non-terminals in the right-hand sides of A, avoiding duplicates. The general left-corner predictor acts when an inactive (completed) edge for a non-terminal A is added at a certain node; it then adds empty active edges for non-terminals that have A as their first non-terminal in their right-hand sides.
Both can be improved by 1. making sure that the added edge has a chance of leading to completion (selectivity); 2. incorporating immediately the non-terminal just recognized (Kilbury); 3. filtering in top-down information. In all tests the selective top-down-filtered Kilbury left-corner predictor clearly outperformed the others.

28. **Rus**, Teodor. Parsing languages by pattern matching. *IEEE Trans. Softw. Eng.*, 14(4):498–511, April 1988. Considers "algebraic grammars" only: there is at least one terminal between each pair of non-terminals in any right-hand side. The rules of the grammar are ordered in "layers", each layer containing only rules whose right-hand sides contain only non-terminals defined in the same or lower layers. On the basis of these layers, very simple contexts are computed for each right-hand side, resulting in an ordered set of patterns. The input is then parsed by repeated application of the patterns in each layer, starting with the bottom one, using fast string matching.
All this is embedded in a system that simultaneously manages abstract semantics. Difficult to read due to unusual terminology.

29. **Kruseman Aretz**, F. E. J. A new approach to Earley's parsing algorithm. *Sci. Comput. Progr.*, 12(2):105–121, July 1989. Starting point is a CYK table filled with Earley items, i.e., a tabular implementation of the Earley algorithm. Rather than implementing the table, two arrays are used, both indexed with the position i in the input. The elements of the first array, D_i, are the mappings from a non-terminal X to the set of all Earley items that have the dot in front of X at position i. The elements of the second array, E_i, are the sets of all reduce items at i. Considerable math is used to derive recurrence relations between the two, leading to a very efficient evaluation order. The data structures are then extended to produce parse trees. Full implementations are given.

30. **Voisin**, Frédéric. and Raoult, Jean-Claude. A new, bottom-up, general parsing algorithm. *BIGRE Bulletin*, 70:221–235, Sept. 1990. Modifies Earley's algorithm by 1. maintaining items of the form α rather than $A \rightarrow \alpha B \bullet \beta$, which eliminates the top-down component and thus the predictive power, and 2. restoring some of that predictive power by predicting items α for each rule in the grammar $A \rightarrow \alpha\beta$ for which the input token is in FIRST(A). This is useful for the special application the authors have, a parser for a language with extensive user-definable operators.

31. **Lang**, Bernard. Towards a uniform formal framework for parsing. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, pages 153–171. Kluwer Academic Publ., Boston, 1991. The paper consists of two disjoint papers. The first concerns the equivalence of grammars and parse forests; the second presents a Logical PushDown Automaton.
In tree-sharing parsers, parsing an (ambiguous) sentence S yields a parse forest. If we label each node in this forest with a unique name, we can consider each node to be a rule in a CF grammar. The node labeled N describes one alternative for the non-terminal N and if p outgoing arrows leave the node, N has p alternatives. This grammar produces exactly one sentence, S. If S contains wild-card tokens, the grammar will produce all sentences in the original grammar that match S. In fact, if we parse the sentence Σ^* in which Σ matches any token, we get a parse forest that is equivalent to the original grammar. Parsing of unambiguous, ambiguous and incomplete sentences alike is viewed as constructing a grammar that produces exactly the singleton, multi-set and infinite set of derivations that produce the members of the input set. No such parsing algorithm is given, but the reader of the paper is referred to Billot and Lang [164], and to Lang [210].
Prolog-like programs can be seen as grammars the non-terminals of which are predicates with arguments, i.e. Horn clauses. Such programs are written as Definite Clause programs. To operate these programs as solution-producing grammars, a Logical PushDown Automaton LPDA is introduced, which uses Earley deduction in a technique similar to that of Pereira and Warren [368]. In this way, a deduction mechanism is obtained that is shown to terminate on a Definite Clause Grammar on which simple depth-first resolution would loop. The conversion from DC program to a set of Floyd-like productions for the LPDA is described in full, and so is the LPDA itself.

32. **Leo**, Joop M. I. M. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoret. Comput. Sci.*, 82:165–176, 1991. Earley parsing of right-recursive LR(k) grammars will need time and space of $O(n^2)$, for the build-up of the final reductions. This build-up is prevented through the introduction of "transitive items", which store right-recursion information. A proof is given that the resulting algorithm is linear in time and space for every LR-regular grammar. The algorithm also defends itself against hidden right recursion.

33. **Nederhof**, M.-J. An optimal tabular parsing algorithm. In *32nd Annual Meeting of the Association for Computational Linguistics*, pages 117–124, June 1994. Like chart parsing, the various LR parsing methods can be characterized by the way they infer new items from old ones. In this paper, four such characterizations are given: for LC parsing, for predictive LR, for regular right part grammars, called "extended LR (ELR) grammars here, and for "Common-Prefix" parsing. For Common-Prefix see Voisin and Raoult [30]. Each of these is then expressed as a tabular parsing algorithm, and their properties are compared. ELR appears the best compromise for power and efficiency.

34. **Rytter**, W. Context-free recognition via shortest-path computation: A version of Valiant's algorithm. *Theoret. Comput. Sci.*, 143(2):343–352, 1995. The multiplication

and addition in the formula for distance in a weighted graph are redefined so that a shortest distance through a specific weighted lattice derived from the grammar and the input corresponds to a parsing. This allows advances in shortest-path computation (e.g. parallelization) to be exploited for parsing. The paper includes a proof that the algorithm is formally equivalent to Valiant's.

35. **McLean**, Philippe and Horspool, R. Nigel. A faster Earley parser. In Tibor Gyimóthy, editor, *Compiler Construction: 6th International Conference, CC'96*, volume 1060 of *Lecture Notes in Computer Science*, pages 281–293, New York, 1996. Springer-Verlag. The items in an Earley set can be grouped into subsets, such that each subset corresponds to an LR state. This is utilized to speed up the Earley algorithm. Speed-up factors of 10 to 15 are obtained, and memory usage is reduced by half.

36. **Johnstone**, Adrian and Scott, Elizabeth. Generalized recursive-descent parsing and follow-determinism. In Kai Koskimies, editor, *Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 16–30, Lisbon, 1998. Springer. The routine generated for a non-terminal A returns a set of lengths of input segments starting at the current position and matching A, rather than just a Boolean saying match or no match. This gives a parser that is efficient on LL(1) and non-left-recursive LR(1). Next it is made more efficient by using FIRST sets. This parser is implemented under the name *GRDP*, for "Generalised Recursive Descent Parser". It yields all parses; but can be asked to act deterministically. It then uses *FOLLOW-determinism*, in which the length is chosen whose segment is followed by a token from $\text{FOLLOW}_1(A)$; the grammar must be such that only one length qualifies.

37. **Aycock**, John and Horspool, R. Nigel. Directly-executable Earley parsing. In *Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 229–243, 2001. Code segments are generated for all actions possible on each possible Earley item, and these segments are linked together into an Earley parser using a threaded-code technique, but parent pointer manipulations are cumbersome. To remedy this, the items are grouped in the states of a split LR(0) automaton, in which each traditional LR(0) state is split in two states, one containing the items in which the dot is at the beginning (the "non-kernel" state), and one which contains the rest. The parent pointers of the non-kernel states are all equal, which simplifies implementation. The resulting parser is 2 to 3 times faster than a standard implementation of the Earley parser.

38. **Aycock**, John and Horspool, R. Nigel. Practical Earley parsing. *Computer J.*, 45(6):620–630, 2002. Empty productions are the reason for the Predictor/Completer loop in an Earley parser, but the loop can be avoided by having the Predictor also predict items of the form $A \rightarrow \alpha B \bullet \beta$ for $\bullet B$ if B is nullable. Effectively the ε is propagated by the Predictor. The nullable non-terminals are found by preprocessing the grammar. The Earley item sets are collected in an "LR(0)ε" automaton. The states of this automaton are then split as described by Aycock and Horspool [37]. A third transformation is required to allow convenient reconstruction of the parse trees.

39. **Lee**, Lillian. Fast context-free grammar parsing requires fast Boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002. The author proves that if we can do parsing in $O(n^{3-\varepsilon})$, we can do Boolean matrix multiplication in $O(n^{3-\varepsilon/3})$. To convert a given matrix multiplication of A and B into a parsing problem we start with a string w of a length that depends only on the size of the matrices; all tokens in w are different. For each non-zero element of A we create a new non-terminal $A_{i,j} \rightarrow w_p W w_q$, where w_p and w_q are judiciously chosen tokens from w and W produces any non-empty string of tokens from w; likewise for B. The resulting matrix C is grammaticalized by rules $C_{p,q} \rightarrow A_{p,r} B_{r,q}$, which implements the (Boolean) multiplication. Occurrences of $C_{p,q}$ are now attempted to be recognized in w by having start symbol rules $S \rightarrow W C_{p,q} W$. The resulting grammar is highly ambiguous, and when we parse w with it, we obtain a parse forest. If the node for $C_{p,q}$ is present in it, the bit $C_{p,q}$ in the resulting matrix is on.

40. **Nederhof**, Mark-Jan and Satta, Giorgio. Tabular parsing. In *Formal Languages and Applications*, volume 148 of *Studies in Fuzziness and Soft Computing*, pages 529–549. Springer, April 2004. Tutorial on tabular parsing for push-down automata, Earley parsers, CYK and non-deterministic LR parsing. The construction of parse trees is also covered.

18.1.3 LL Parsing

41. **Lucas**, P. Die Strukturanalyse von Formelnübersetzern / analysis of the structure of formula translators. *Elektronische Rechenanlagen*, 3(11.4):159–167, 1961, (in German).
Carefully reasoned derivation of a parser and translator from the corresponding grammar. Two types of "syntactic variable definitions" (= grammar rules) are distinguished: "enumerating definitions", of the form $N \rightarrow A_1|A_2|\cdots$, and "concatenating definitions", of the form $N \rightarrow A_1A_2\cdots$; here N is a non-terminal and the A_i are all terminals or non-terminals. Additionally "combined definitions" are allowed, but they are tacitly decomposed into enumerating and concatenating definitions when the need arises. Each "syntactic variable" (= non-terminal) can be "explicitly defined", in which case the definition does not refer to itself, or "recursively defined", in which case it does.
For each non-terminal N two pieces of code are created: an identification routine and a translation routine. The identification routine tests if the next input token can start a terminal production of N, and the translation routine parses and translates it. The identification routines are produced by inspecting the grammar; the translation routines are created from templates, as follows. The translation routine for an enumerating definition is

```
if can.start.A1 then translate.A1 else
if can.start.A2 then translate.A2 else
... else report.error
```

The translation routine for a concatenating definition is

```
if can.start.A1 then translate.A1 else report.error;
if can.start.A2 then translate.A2 else report.error;
...
```

Each translation routine can have local variables and produces an output parameter, which can be used for code generation by the caller; all these variables are allocated statically (as global memory locations). These routines are given by flowcharts, although the author recognizes that they could be expressed in ALGOL 60.
The flowcharts are connected into one big flow chart, except that a translation routine for a recursive non-terminal starts with code that stores its previous return address and local variables on a stack, and ends with code that restores them. Since the number of local variables vary from routine to routine, the stack entries are of unequal size; such stack entries are called "drawers". No hint is given that the recursion of ALGOL 60 could be used for these purposes. Special flowchart templates are given for directly left- and right-recursive non-terminals, which transform the recursion into iteration.
For the system to work the grammar must obey requirements that are similar to LL(1), although the special treatment of direct left recursion alleviates them somewhat. These requirements are mentioned but not analysed. Nothing is said about nullable non-terminals.
The method is compared to the PDA technique of Samelson and Bauer, [112]. The two methods are recognized as equivalent, but the method presented here lends itself better for hand optimization and code insertion.
This is probably the first description of recursive descent parsing. The author states that four papers explaining similar techniques appeared after the paper was written but before it was printed. That depends on the exact definition of recursive descent: of the four only Grau [332] shows how to generate code for the routines, i.e., to use compiled recursive descent. The others interpret the grammar.

42. **Kurki-Suonio**, R. Notes on top-down languages. *BIT*, 9:225–238, 1969. Gives several variants of the LL(k) condition. Also demonstrates the existence of an LL(k) language which is not LL($k-1$).

43. **Knuth**, Donald E. Top-down syntax analysis. *Acta Inform.*, 1:79–110, 1971. A *Parsing Machine* (PM) is defined, which is effectively a set of mutually recursive Boolean functions which absorb input if they succeed and absorb nothing if they fail. Properties of the languages accepted by PMs are examined. This leads to CF grammars, dependency graphs, the null string problem, back-up, LL(k), follow function, LL(1), s-languages and a comparison of top-down versus bottom-up parsing. The author is one of the few scientists who provides insight in their thinking process.

44. **Jarzabek**, S. and Krawczyk, T. LL-regular grammars. *Inform. Process. Lett.*, 4(2):31–37, 1975. Introduces LL-regular (*LLR*) grammars: for every rule $A \rightarrow \alpha_1|\cdots|\alpha_n$, a partition $(R_1,\ldots,R_n)$ of disjoint regular sets must be given such that the rest of the input sentence is a member of exactly one of these sets. A parser can then be constructed by creating finite-state automata for these sets, and letting these finite state automata determine the next prediction.

45. **Nijholt**, A. On the parsing of LL-regular grammars. In A. Mazurkiewicz, editor, *Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 446–452. Springer-Verlag, Berlin, 1976. Derives a parsing algorithm for LL-regular grammars with a regular pre-scan from right to left that leaves markers, and a subsequent scan which consists of an LL(1)-like parser.

46. **Lewi**, J., Vlaminck, K. de, Huens, J., and Huybrechts, M. The ELL(1) parser generator and the error-recovery mechanism. *Acta Inform.*, 10:209–228, 1978. See same paper [298].

47. **Poplawski**, D. A. On LL-regular grammars. *J. Comput. Syst. Sci.*, 18:218–227, 1979. Presents proof that, given a regular partition, it is decidable whether a grammar is LL-regular with respect to this partition; it is undecidable whether or not such a regular partition exists. The paper then discusses a two-pass parser; the first pass works from right to left, marking each terminal with an indication of the partition that the rest of the sentence belongs to. The second pass then uses these indications for its predictions.

48. **Nijholt**, A. LL-regular grammars. *Intern. J. Comput. Math.*, A8:303–318, 1980. This paper discusses strong-LL-regular grammars, which are a subset of the LL-regular grammars, exactly as the strong-LL(k) grammars are a subset of the LL(k) grammars, and derives some properties.

49. **Heckmann**, Reinhold. An efficient ELL(1)-parser generator. *Acta Inform.*, 23:127–148, 1986. The problem of parsing with an ELL(1) grammar is reduced to finding various FIRST and FOLLOW sets. Theorems about these sets are derived and very efficient algorithms for their computation are supplied.

50. **Barnard**, David T. and Cordy, James R. Automatically generating SL parsers from LL(1) grammars. *Comput. Lang.*, 14(2):93–98, 1989. For SL see Barnard and Cordy [265]. SL seems ideally suited for implementing LL(1) parsers, were it not that the choice action absorbs the input token on which the choice is made. This effectively prevents look-ahead, and means that when a routine for a non-terminal A is called, its first token has already been absorbed. A scheme is suggested that will replace the routine for parsing A by a routine for parsing A minus its first token. So the technique converts the grammar to simple LL(1).

51. **Parr**, Terence J. and Quong, Russell W. LL and LR translators need $k > 1$ lookahead. *ACM SIGPLAN Notices*, 31(2):27–34, Feb. 1996. Gives realistic examples of frequent programming language constructs in which $k = 1$ fails. Since $k > 1$ is very expensive, the authors introduce linear-approximate LL(k) with $k > 1$, in which for each look-ahead situation S separate values FIRST(S), SECOND(S), ..., are computed, which needs $t \times k$ space for t equal to the number of different tokens, rather than $\text{FIRST}_k(S)$, which requires t^k. This may weaken the parser since originally differing look-ahead sets like ab, cd and ad, cb both collapse to $[ac][bd]$, but usually works out OK.

18.1.4 LR Parsing

52. **Knuth**, D. E. On the translation of languages from left to right. *Inform. Control*, 8:607–639, 1965. This is the original paper on LR(k). It defines the notion as an abstract property of a grammar and gives two tests for LR(k). The first works by constructing for the grammar a regular grammar which generates all possible already reduced parts (= stacks) plus their look-aheads; if this grammar has the property that none of its words is a prefix to another of its words, the original grammar was LR(k). The second consists of implicitly constructing all possible item sets (= states)

and testing for conflicts. Since none of this is intended to yield a reasonable parsing algorithm, notation and terminology differs from that in later papers on the subject. Several theorems concerning LR(k) grammars are given and proved.

53. **Korenjak**, A. J. A practical method for constructing LR(k) processors. *Commun. ACM*, 12(11):613–623, Nov. 1969. The huge LR(1) parsing table is partitioned as follows. A nonterminal Z is chosen judiciously from the grammar, and two grammars are constructed, G_0, in which Z is considered to be a terminal symbol, and G_1, which is the grammar for Z (i.e. which has Z as the start symbol). If both grammars are LR(1) and moreover a master LR(1) parser can be constructed that controls the switching back and forth between G_0 and G_1, the parser construction succeeds (and the original grammar was LR(1) too). The three resulting tables together are much smaller than the LR(1) table for the original grammar. It is also possible to chose a set of nonterminals $Z_1 \cdots Z_n$ and apply a variant of the above technique.

54. **DeRemer**, Franklin L. Simple LR(k) grammars. *Commun. ACM*, 14(7):453–460, July 1971. SLR(k) explained by its inventor. Several suggestions are made on how to modify the method; use a possibly different k for each state; use possibly different lengths for each look-ahead string. The relation to Korenjak's approach [53] is also discussed.

55. **Anderson**, T. *Syntactic Analysis of LR(k) Languages*. PhD thesis, Technical report, University of Newcastle upon Tyne, Newcastle upon Tyne, 1972. [Note: This is one of the few papers we have not been able to access; the following is the author's abstract.] A method of syntactic analysis, termed LA(m)LR(k), is discussed theoretically. Knuth's LR(k) algorithm [52] is included as the special case $m = k$. A simpler variant, SLA(m)LR(k), is also described, which in the case SLA(k)LR(0) is equivalent to the SLR(k) algorithm as defined by DeRemer [54]. Both variants have the LR(k) property of immediate detection of syntactic errors.
The case $m = 1, k = 0$ is examined in detail, when the methods provide a practical parsing technique of greater generality than precedence methods in current use. A formal comparison is made with the weak precedence algorithm.
The implementation of an SLA(1)LR(0) parser (SLR) is described, involving numerous space and time optimizations. Of importance is a technique for bypassing unnecessary steps in a syntactic derivation. Direct comparisons are made, primarily with the simple precedence parser of the highly efficient Stanford ALGOL W compiler, and confirm the practical feasibility of the SLR parser.

56. **Anderson**, T., Eve, J., and Horning, J. J. Efficient LR(1) parsers. *Acta Inform.*, 2:12–39, 1973. Coherent explanation of SLR(1), LALR(1), elimination of unit rules and table compression, with good advice.

57. **Čulik**, II, Karel and Cohen, Rina. LR-regular grammars: An extension of LR(k) grammars. *J. Comput. Syst. Sci.*, 7:66–96, 1973. The input is scanned from right to left by a FS automaton which records its state at each position. Next this sequence of states is parsed from left to right using an LR(0) parser. If such a FS automaton and LR(0) parser exist, the grammar is LR-regular. The authors conjecture, however, that it is unsolvable to construct this automaton and parser. Examples are given of cases in which the problem can be solved.

58. **LaLonde**, Wilf R. Regular right part grammars and their parsers. *Commun. ACM*, 20(10):731–741, Oct. 1977. The notion of regular right part grammars and its advantages are described in detail. A parser is proposed that does LR(k) parsing to find the right end of the handle and then, using different parts of the same table, scans the stack backwards using a look-ahead (to the left!) of m symbols to find the left end; this is called *LR(m,k)*. The corresponding parse table construction algorithm is given by LaLonde [59].

59. **LaLonde**, W. R. Constructing LR parsers for regular right part grammars. *Acta Inform.*, 11:177–193, 1979. Describes the algorithms for the regular right part parsing technique explained by LaLonde [58]. The back scan is performed using so-called *read-back tables*. Compression techniques for these tables are given.

60. **Baker**, Theodore P. Extending look-ahead for LR parsers. *J. Comput. Syst. Sci.*, 22(2):243–259, 1981. A FS automaton is derived from the LR automaton as follows: upon a

reduce to A the automaton moves to all states that have an incoming arc marked A. This automaton is used for analysing the look-ahead as in an LR-regular parser (Čulik, II and Cohen [57]).

61. **Kristensen**, Bent Bruun and Madsen, Ole Lehrmann. Methods for computing LALR(k) lookahead. *ACM Trans. Prog. Lang. Syst.*, 3(1):60–82, Jan. 1981. The LALR(k) look-ahead sets are seen as the solution to a set of equations, which are solved by recursive traversal of the LR(0) automaton. Full algorithms plus proofs are given.

62. **LaLonde**, Wilf R. The construction of stack-controlling LR parsers for regular right part grammars. *ACM Trans. Prog. Lang. Syst.*, 3(2):168–206, April 1981. Traditional LR parsers shift each input token onto the stack; often, this shift could be replaced by a state transition, indicating that the shift has taken place. Such a parser is called a *stack-controlling LR parser*, and will do finite-state recognition without stack manipulation whenever possible. Algorithms for the construction of stack-controlling LR parse tables are given. The paper is complicated by the fact that the new feature is introduced not in a traditional LR parser, but in an LR parser for regular right parts (for which see LaLonde [58]).

63. **DeRemer**, Frank L. and Pennello, Thomas J. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Prog. Lang. Syst.*, 4(4):615–649, Oct. 1982. Rather than starting from an LR(1) automaton and collapsing it to obtain an LALR(1) automaton, the authors start from an LR(0) automaton and compute the LALR(1) look-aheads from there, taking into account that look-aheads are meaningful for reduce items only. For each reduce item $A \rightarrow \alpha\bullet$ we search back in the LR(0) automaton to find all places P where it could originate from, for each of these places we find the places Q that can be reached by a shift over A from P, and from each of these places we look forward in the LR(0) automaton to determine what the next token in the input could be. The set of all these tokens is the LALR(1) look-ahead set of the original reduce item. The process is complicated by the presence of ε-productions.
The computation is performed by four linear sweeps over the LR(0) automaton, set up so that they can be implemented by transitive closure algorithms based on strongly connected components, which are very efficient.
Care must be taken to perform the above computations in the right order; otherwise look-ahead sets may be combined too early resulting in "Not Quite LALR(1)", NQLALR(1), which is shown to be inadequate.
The debugging of non-LALR(1) grammars is also treated.

64. **Heilbrunner**, S. Truly prefix-correct chain-free LR(1) parsers. *Acta Inform.*, 22(5):499–536, 1985. A unit-free LR(1) parser generator algorithm, rigorously proven correct.

65. **Park**, Joseph C. H., Choe, K.-M., and Chang, C.-H. A new analysis of LALR formalisms. *ACM Trans. Prog. Lang. Syst.*, 7(1):159–175, Jan. 1985. The recursive closure operator *CLOSURE* of Kristensen and Madsen [61] is abstracted to an iterative δ-operator such that $CLOSURE \equiv \delta^*$. This operator allows the formal derivation of four algorithms for the construction of LALR look-ahead sets, including an improved version of the relations algorithm of DeRemer and Pennello [63]. See Park and Choe [73] for an update.

66. **Ukkonen**, Esko. Upper bounds on the size of LR(k) parsers. *Inform. Process. Lett.*, 20(2):99–105, Feb. 1985. Upper bounds for the number of states of an LR(k) parser are given for several types of grammars.

67. **Al-Hussaini**, A. M. M. and Stone, R. G. Yet another storage technique for LR parsing tables. *Softw. Pract. Exper.*, 16(4):389–401, 1986. Excellent introduction to LR table compression in general. The *submatrix technique* introduced in this paper partitions the rows into a number of submatrices, the rows of each of which are similar enough to allow drastic compressing. The access cost is $O(1)$. A heuristic partitioning algorithm is given.

68. **Ives**, Fred. Unifying view of recent LALR(1) lookahead set algorithms. *ACM SIGPLAN Notices*, 21(7):131–135, July 1986. A common formalism is given in which the LALR(1) look-ahead set construction algorithms of DeRemer and Pennello [63], Park, Choe and Chang [65] and the author can be expressed. See also Park and Choe [73].

69. **Nakata**, Ikuo and Sassa, Masataka. Generation of efficient LALR parsers for regular right part grammars. *Acta Inform.*, 23:149–162, 1986. The stack of an LALR(1) parser is augmented with a set of special markers that indicate the start of a right-hand side; adding such a marker during the shift is called a *stack shift*. Consequently there can now be a shift/stack-shift conflict, abbreviated to *stacking conflict*. The stack-shift is given preference and any superfluous markers are eliminated during the reduction. Full algorithms are given.

70. **Pennello**, Thomas J. Very fast LR parsing. *ACM SIGPLAN Notices*, 21(7):145–151, July 1986. The tables and driver of a traditional LALR(1) parser are replaced by assembler code performing linear search for small fan-out, binary search for medium and a computed jump for large fan-out. This modification gained a factor of 6 in speed at the expense of a factor 2 in size.

71. **Chapman**, Nigel P. *LR Parsing: Theory and Practice*. Cambridge University Press, New York, NY, 1987. Detailed treatment of the title subject. Highly recommended for anybody who wants to acquire in-depth knowledge about LR parsing. Good on size of parse tables and attribute grammars.

72. **Ives**, Fred. Response to remarks on recent algorithms for LALR lookahead sets. *ACM SIGPLAN Notices*, 22(8):99–104, Aug. 1987. Remarks by Park and Choe [73] are refuted and a new algorithm is presented that is significantly better than that of Park, Choe and Chang [65] and that previously presented by Ives [68].

73. **Park**, Joseph C. H. and Choe, Kwang-Moo. Remarks on recent algorithms for LALR lookahead sets. *ACM SIGPLAN Notices*, 22(4):30–32, April 1987. Careful analysis of the differences between the algorithms of Park, Choe and Chang [65] and Ives [68]. See also Ives [72].

74. **Sassa**, Masataka and Nakata, Ikuo. A simple realization of LR-parsers for regular right part grammars. *Inform. Process. Lett.*, 24(2):113–120, Jan. 1987. For each item in each state on the parse stack of an LR parser, a counter is kept indicating how many preceding symbols on the stack are covered by the recognized part in the item. Upon reduction, the counter of the reducing item tells us how many symbols to unstack. The manipulation rules for the counters are simple. The counters are stored in short arrays, one array for each state on the stack.

75. **Bermudez**, Manuel E. and Schimpf, Karl M. On the (non-)relationship between SLR(1) and NQLALR(1) grammars. *ACM Trans. Prog. Lang. Syst.*, 10(2):338–342, April 1988. Shows a grammar that is SLR(1) but not NQLALR(1).

76. **Bermudez**, Manuel E. and Schimpf, Karl M. A general model for fixed look-ahead LR parsers. *Intern. J. Comput. Math.*, 24(3+4):237–271, 1988. Extends the DeRemer and Pennello [63] algorithm to LALR(k), NQLALR(k) and SLR(k). Also defines NQSLR(k), Not-Quite SLR, in which a too simple definition of FOLLOW_k is used. The difference only shows up for $k \geq 2$, and is similar to the difference in look-ahead between full-LL(k) and strong-LL(k). Suppose for $k = 2$ we have the grammar **S→ApBq**, **S→Br**, **A→a**, **B→**ε, and we compute FOLLOW_2(**A**). Then the NQSLR algorithm computes it as FIRST_1(**B**) plus FOLLOW_1(**B**) if **B** produces ε. Since FOLLOW_1(**B**)={**q**,**r**}, this yields the set {**pq**,**pr**}; but the sequence **pr** cannot occur in any input. The authors give an example where such an unjustified look-ahead prevents parser construction.

77. **Kruseman Aretz**, F. E. J. On a recursive ascent parser. *Inform. Process. Lett.*, 29(4):201–206, Nov. 1988. Each state in an LR automaton is implemented as a subroutine. A shift calls that subroutine. A reduce to X is effected as follows. X and its length n are stored in global variables; all subroutines are rigged to decrement n and return as long as $n > 0$, and to call the proper GOTO state of X when n hits 0. This avoids the explicit stack manipulation of Roberts [78].

78. **Roberts**, George H. Recursive ascent: An LR analog to recursive descent. *ACM SIGPLAN Notices*, 23(8):23–29, Aug. 1988. Each LR state is represented by a subroutine. The shift is implemented as a subroutine call; the reduction is followed by a subroutine return possibly preceded by a return stack adjustment. The latter prevents the generation of genuine subroutines

since it requires explicit return stack manipulation. A small and more or less readable LR(0) parser is shown, in which conflicts are resolved by means of the order in which certain tests are done, like in a recursive descent parser.

79. **Bermudez**, Manuel E. and Logothetis, George. Simple computation of LALR(1) look-ahead sets. *Inform. Process. Lett.*, 31(5):233–238, 1989. The original LALR(1) grammar is replaced by a not much bigger grammar that has been made to incorporate the necessary state splitting through a simple transformation. The SLR(1) automaton of this grammar is the LALR(1) automaton of the original grammar.

80. **Horspool**, R. Nigel. ILALR: An incremental generator of LALR(1) parsers. In D. Hammer, editor, *Compiler Compilers and High-Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 128–136. Springer-Verlag, Berlin, 1989. Grammar rules are checked as they are typed in. To this end, LALR(1) parse tables are kept and continually updated. When the user interactively adds a new rule, the sets FIRST and NULLABLE are recomputed and algorithms are given to distribute the consequences of possible changes over the LR(0) and look-ahead sets. Some serious problems are reported and practical solutions are given.

81. **Roberts**, George H. Another note on recursive ascent. *Inform. Process. Lett.*, 32(5):263–266, 1989. The fast parsing methods of Pennello [70], Kruseman Aretz [77] and Roberts [78] are compared. A special-purpose optimizing compiler can select the appropriate technique for each state.

82. **Bermudez**, Manuel E. and Schimpf, Karl M. Practical arbitrary lookahead LR parsing. *J. Comput. Syst. Sci.*, 41(2):230–250, Oct. 1990. Refines the extended-LR parser of Baker [60] by constructing a FS automaton for each conflict state q as follows. Starting from q and looking backwards in the LR(0) automaton, all top-of-stack segments of length m are constructed that have q on the top. These segments define a regular language R which is a superset of the possible continuations of the input (which are determined by the entire stack). Also each decision made by the LR(0) automaton to resolve the conflict in q defines a regular language, each a subset of R. If these languages are disjunct, we can decide which decision to take by scanning ahead. Scanning ahead is done using an automaton derived from q and the LR(0) automaton. Grammars for which parsers can be constructed by this technique are called LAR(m). The technique can handle some non-LR(k) grammars.

83. **Heering**, J., Klint, P., and Rekers, J. Incremental generation of parsers. *IEEE Trans. Softw. Eng.*, 16(12):1344–1351, 1990. In a very unconventional approach to parser generation, the initial information for an LR(0) parser consists of the grammar only. As parsing progresses, more and more entries of the LR(0) table (actually a graph) become required and are constructed on the fly. LR(0) inadequacies are resolved using GLR parsing. All this greatly facilitates handling (dynamic) changes to the grammar.

84. **Horspool**, R. N. Incremental generation of LR parsers. *Comput. Lang.*, 15(4):205–233, 1990. The principles and usefulness of incremental parser generation are argued. The LR parse table construction process is started with initial items $S_S \rightarrow \bullet\#_N N\#_N$ for each non-terminal N in the grammar, where the $\#_N$s are N-specific delimiters. The result is a parse table in which any non-terminal can be the start symbol. When N is modified, the item $S_S \rightarrow \bullet\#_N N\#_N$ is followed throughout the parser, updates are made, and the table is cleaned of unreachable items. Deletion is handled similarly. The algorithms are outlined.
A proof is given that modifying a single rule in a grammar of n rules can cause $O(2^n)$ items to be modified, so the process is fundamentally exponential. In practice, it turns out that incremental recomputing is in the order of 10 times cheaper than complete recomputing.

85. **Horspool**, R. Nigel and Whitney, Michael. Even faster LR parsing. *Softw. Pract. Exper.*, 20(6):515–535, June 1990. Generates directly executable code. Starts from naive code and then applies a series of optimizations: 1. Push non-terminals only; this causes some trouble in deciding what to pop upon and reduce and it may even introduce a pop-count conflict. 2. Combine the reduce

and the subsequent goto into one optimized piece of code. 3. Turn right recursion into left recursion to avoid stacking and generate code to undo the damage. 4. Eliminate unit rules.

86. **McKenzie**, B. J. LR parsing of CFGs with restrictions. *Softw. Pract. Exper.*, 20(8):823–832, Aug. 1990. It is often useful to specify semantic restrictions at certain points in the right-hand sides of grammar rules; these restrictions can serve to check the semantic correctness of the input and/or to disambiguate the grammar. Conceptually these restrictions are associated with marker non-terminals, which produce ε and upon reduction test the restriction. This causes lots of conflicts in the LR parser; rather than have the LR parser generator solve them in the usual fashion, they are solved at parse time by calling the restriction-testing routines. If no test routine succeeds, there is an error in the input; if one succeeds, the parser knows what to do; and if more than one succeeds, there is a grammar error, which can be dealt with by having a default (use the textually first restriction, for example), or by giving an error message. Many examples, no explicit code. It would seem the system can also be used to implement dynamic conflict resolvers.

87. **Roberts**, George H. From recursive ascent to recursive descent: via compiler optimizations. *ACM SIGPLAN Notices*, 25(4):83–89, April 1990. Shows a number of code transformations that will turn an LR(1) recursive ascent parser (see Roberts [78, 81]) for an LL(1) grammar into a recursive descent parser.

88. **Charles**, Phillippe. *A Practical Method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, Technical report, NYU, Feb. 1991. Addresses various issues in LALR parsing: 1. Gives an in-depth overview of LALR parsing algorithms. 2. Modifies DeRemer and Pennello's algorithm [63] to adapt the length of the lookaheads to the needs of the states. 3. Gives an improved version of Burke and Fisher' automatic LR error recovery mechanism [317], for which see [319]. 4. Existing table compression methods are tuned to LALR tables. Explicit algorithms are given.

89. **Fortes Gálvez**, José. Generating LR(1) parsers of small size. In U. Kastens and P. Pfahler, editors, *Compiler Construction, 4th International Conference, CC'92*, volume 641 of *Lecture Notes in Computer Science*, pages 16–29. Springer-Verlag, Oct. 1992. Actually, reverse LR(1) parsers are constructed, as follows. The stack is the same as for the normal LR(1) parser, except that no states are recorded, so the stack consists of non-terminals and terminals only. When faced with the problem of whether to shift or to reduce, the stack is analysed from the top downward, rather than from the bottom upward. Since the top region of the stack contains more immediately relevant information than the bottom region, the above analysis will usually come up with an answer pretty quickly.
The analysis can be done using an FSA, starting with the look-ahead token. An algorithm to construct this FSA is described informally, and a proof is given that it has the full LR(1) parsing power. The resulting automaton is about 1/3 the size of the *yacc* automaton, so it is even smaller than the LALR(1) automaton.

90. **Shin**, Heung-Chul and Choe, Kwang-Moo. An improved LALR(*k*) parser generation for regular right part grammars. *Inform. Process. Lett.*, 47(3):123–129, 1993. Improves the algorithm of Nakata and Sassa [69] by restricting the algorithm to kernel items only.

91. **Fortes Gálvez**, José. A note on a proposed LALR parser for extended context-free grammars. *Inform. Process. Lett.*, 50(6):303–305, June 1994. Shows that the algorithm of Shin and Choe [90] is incorrect by giving a counterexample.

92. **Fortes Gálvez**, José. A practical small LR parser with action decision through minimal stack suffix scanning. In *Developments in Language Theory II*, pages 460–465, Singapore, 1995. World Scientific. Theory of and explicit algorithms for DR parsing.

93. **Seyfarth**, Benjamin R. and Bermudez, Manuel E. Suffix languages in LR parsing. *Intern. J. Comput. Math.*, A-55(3-4):135–154, 1995. An in-depth analysis of the set of strings that can follow a state in a non-deterministic LR(0) automation (= an item in the deter-

ministic one) is given and used to derive all known LR parsing algorithms. Based on first author's thesis.

94. **Nederhof**, Mark-Jan and Sarbo, Janos J. Increasing the applicability of LR parsing. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*, pages 35–58. Kluwer Academic Publishers, Dordrecht, 1996. ε-reductions are incorporated in the LR items, resulting in ε-LR parsing. Now the stack contains only non-terminals that correspond to non-empty segments of the input; it may be necessary to examine the stack to find out exactly which reduction to do. ε-LR parsing has two advantages: more grammars are ε-LR than LR; and non-deterministic ε-LR tables will never make the original Tomita algorithm [162] loop, thus providing an alternative way to do GLR parsing on arbitrary CF grammars, in addition to Nozohoor-Farshi's method [167].

95. **Fortes Gálvez**, José. *A Discriminating-Reverse Approach To LR(k) Parsing*. PhD thesis, Technical report, Univesité de Nice-Sophia Antipolis, Nice, France, 1998. Existing parsing techniques are explained and evaluated for convenience and memory use. Several available implementations are also discussed. The convenience of full LR(1), LR(2), etc. parsing with minimal memory use is obtained with DR parsing. The DR(0) and DR(1) versions are discussed in detail, and measurements are provided; theory of DR(k) is given. Algorithms for ambiguous grammars are also presented.

96. **Bertsch**, Eberhard and Nederhof, Mark-Jan. Regular closure of deterministic languages. *SIAM J. Computing*, 29(1):81–102, 1999. A *meta-deterministic language* is a language expressed by a regular expression the elements of which are LR(0) languages. Every LR(k) language is meta-deterministic, i.e., can be formed as a regular sequence of LR(0) languages. Using a refined form of the technique of Bertsch [215], in which the above regular expression plays the role of the root set grammar, the authors show that meta-deterministic languages can be recognized and parsed in linear time. Many proofs, much theory.

97. **Morimoto**, Shin-Ichi and Sassa, Masataka. Yet another generation of LALR parsers for regular right part grammars. *Acta Inform.*, 37(9):671–697, 2000. To allow determining the extent of the handle of a reduce, markers are pushed on the stack whenever a production could start. For most LALR(1) grammars these allow unique identification of the handle segment at reduce time. For other LALR(1) grammars counters are included in the stack. Complicated theory, but extensive examples given.

98. **Farré**, Jacques and Fortes Gálvez, José. A bounded graph-connect construction for LR-regular parsers. In *Compiler Construction: 10th International Conference, CC 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 244–258. Springer-Verlag, 2001. Detailed description of the constructing of a practical LR-regular parser, consisting of both algorithms and heuristic rules for the development of the look-ahead automata. As an example, such a parser is constructed for a difficult subset of HTML.

99. **Kannapinn**, Sönke. *Eine Rekonstruktion der LR-Theorie zur Elimination von Redundanzen mit Anwendung auf den Bau von ELR-Parsern*. PhD thesis, Technical report, Technische Universität Berlin, Berlin, July 2001, (in German). The thesis consists of two fairly disjunct parts; the first part (100 pages) concerns redundancy in LR parsers, the second (60 pages) designs an LR parser for EBNF, after finding errors in existing publications.
The states in an LR parser hold a lot of redundancy: for example, the top state on the stack is not at all independent of the rest of the stack. This is good for time efficiency but bad for space efficiency. The states in an LR parser serve to answer three questions: 1. whether to shift or to reduce; if to reduce, 2. what rule to use; 3. to what new state to go to after the reduce. In each of these a look-ahead can be taken into account. Any scheme that provides these answers works.
The author proposes various ways to reduce the amount of information carried in the dotted items, and the LR, LALR and SLR states. In each of these cases, the ability to determine the reduce rule suffers, and further stack examination is required to answer question 2 above; this stack examination must be of bounded size, or else the parser is no longer linear-time. Under some of the

modifications, the original power remains, but other classes of grammars also appear: *algemeines LR*, translated by Joachim Durchholz by *Compact LR*, since the literal translation "general LR" is too much like "generalized LR", and ILALR.
In Compact LR, an item $A \rightarrow \alpha \bullet X \beta, t$ in a state of the LR(1) automaton is reduced to $X|u$ where X can be terminal or non-terminal, and u is the immediate look-ahead of X, i.e. the first token of β if it exists, or t if β is absent. The resulting *CLR(1)* automaton collapses considerably; for example, all reduce-only states are now empty (since X is absent) and can be combined. This automaton has the same shift behavior as the LR(1) automaton, but when a reduce is called for, no further information is available from the automaton, and stack examination is required. If the stack examination is of bounded size, the grammar was CLR(1).
The design of the LR state automata is given in great detail, with examples, but the stack examination algorithms are not given explicitly, and no examples are provided. No complete parsing example is given.
Should have been in English.

100. **Scott**, Elizabeth and Johnstone, Adrian. Reducing non-determinism in reduction-modified LR(1) parsers. Technical Report CSD-TR-02-04, Royal Holloway, University of London, Jan. 2002. Theory of the reduction-modified LR(1) parser used in GRMLR parsing (Scott [177]), plus some improvements.

18.1.5 Left-Corner Parsing

This section also covers a number of related top-down non-canonical techniques: production chain, LLP(k), PLR(k), etc. The bottom-up non-canonical techniques are collected in (Web)Section 18.2.2.

101. **Rosenkrantz**, D. J. and Lewis, II, P. M. Deterministic left-corner parsing. In *IEEE Conference Record 11th Annual Symposium on Switching and Automata Theory*, volume 11, pages 139–152, 1970. An LC(k) parser decides the applicability of a rule when it has seen the initial non-terminal of the rule if it has one, plus a look-ahead of k symbols. Identifying the initial non-terminal is done by bottom-up parsing, the rest of the rule is recognized top-down. A canonical LC pushdown machine can be constructed in which the essential entries on the pushdown stack are pairs of non-terminals, one telling what non-terminal has been recognized bottom-up and the other what non-terminal is predicted top-down. As with LL, there is a difference between LC and strong-LC. There is a simple algorithm to convert an LC(k) grammar into LL(k) form; the resulting grammar may be large, though.

102. **Lomet**, David B. Automatic generation of multiple exit parsing subroutines. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 214–231. Springer-Verlag, Berlin, 1974. A *production chain* is a chain of production steps $X_0 \rightarrow X_1\alpha_1$, $X_1 \rightarrow X_2\alpha_2$, ..., $X_{n-1} \rightarrow \mathtt{t}\alpha_n$, with $X_0 \ldots X_n - 1$ non-terminals and `t` a terminal. If the input is known to derive from X_0 and starts with `t`, each production chain from X_0 to `t` is a possible explanation of how `t` was produced. The set of all production chains connecting X_0 to `t` is called a *production expression*. An efficient algorithm for the construction and compression of production expressions is given. Each production expression is then implemented as a subroutine which contains the production expression as a FS automaton.

103. **Demers**, Alan J. Generalized left corner parsing. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 170–182, New York, 1977. ACM. The right-hand side of each rule is required to contain a marker. The part on the left of the marker is the left corner; it is recognized by SLR(1) techniques, the rest by LL(1) techniques. An algorithm is given to determine the first admissible position in each right-hand side for the marker. Note that this is unrelated to the Generalized Left-Corner Parsing of Nederhof [172].

104. **Soisalon-Soininen**, Eljas and Ukkonen, Esko. A method for transforming grammars into LL(k) form. *Acta Inform.*, 12:339–369, 1979. Introduces a subclass of the LR(k) grammars called predictive LR(k) (PLR(k)). The deterministic LC(k) grammars are strictly included in this class, and a grammatical transformation is presented to transform a PLR(k) into an LL(k) grammar. PLR(k) grammars can therefore be parsed with the LL(k) parser of the transformed grammar. A consequence is that the classes of LL(k), LC(k), and PLR(k) languages are identical.

105. **Nederhof**, M.-J. A new top-down parsing algorithm for left-recursive DCGs. In *5th International Symposium on Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 108–122. Springer-Verlag, Aug. 1993. "Cancellation parsing" predicts alternatives for leftmost non-terminals, just as any top-down parser does, but keeps a set of non-terminals that have already been predicted as left corners, and when a duplicate turns up, the process stops. This basically parses the largest left-corner tree with all different non-terminals on the left spine. The original prediction then has to be restarted to see if there is a still larger tree.
It is shown that this is the minimal extension of top-down parsing that can handle left recursion. The parser can be made deterministic by using look-ahead and three increasingly demanding definitions are given, leading to $C(k)$, strong $C(k)$ and severe $C(k)$. It is shown that $LL(k) \subset C(k) \subset LC(k)$ and likewise for the strong variant. Cancellation parsing cannot handle hidden left recursion.
The non-deterministic case is presented as definition-clause grammars, and an algorithm is given to use attributes to aid in handling hidden left recursion. The generation of a non-deterministic cancellation parser requires no analysis of the grammar: each rule can be translated in isolation.
See also Chapter 5 of Nederhof's thesis [156].

106. **Žemlička**, Michal and Král, Jaroslav. Run-time extensible deterministic top-down parsing. *Grammars*, 2(3):283–293, 1999. Easy introduction to "kind" grammars. Basically a grammar is *kind* if it is LL(1) after left-factoring and eliminating left recursion. The paper explains how to perform these processes automatically during parser generation, which results in traditional-looking and easily modifiable recursive descent parsers. The corresponding pushdown automaton is also described.

107. **Žemlička**, Michal. Parsing with oracle. In *Text, Speech and Dialogue*, volume 4188 of *Lecture Notes in Computer Science*, pages 309–316. Springer, 2006. Summary of the definitions of the oracle-enhanced parsing automata from [108]; no examples, no applications. "Oracle" is not a language, as the title suggests, but just "an oracle".

108. **Žemlička**, Michal. *Principles of Kind Parsing*. PhD thesis, Technical report, Charles University, Prague, June 2006. Theory, practice, applications, and a parser for kind grammars [106], extensively explained. The parser is based on an oracle-enhanced PDA. To this end the notion of look-ahead is extended to that of an oracle, which allows great freedom of adaptation and modification. The automated construction of oracles for complicated look-ahead sets is discussed and examples are given.

18.1.6 Precedence and Bounded-Right-Context Parsing

Papers on bounded-context (BC) and bounded-context parsable (BCP), which are non-canonical, can be found in (Web)Section 18.2.2.

109. **Adams**, Eldridge S. and Schlesinger, Stewart I. Simple automatic coding systems. *Commun. ACM*, 1(7):5–9, July 1958. Describes a simple parser for arithmetic expressions: read the entire expression, start at the end, find the first open parenthesis, from there find the first closing parenthesis to the right, translate the isolated parentheses-free expression, replace by result, and repeat until all parentheses are gone. A parentheses-free expression is parsed by distinguishing between one-factor terms and more-than-one-factor terms, but the algorithm is not made explicit.

110. **Wolpe**, Harold. Algorithm for analyzing logical statements to produce a truth function table. *Commun. ACM*, 1(3):4–13, March 1958. The paper describes an algorithm to convert a Boolean expression into a decision table. The expression is first fully parenthesized through a number of substitution rules that represent the priorities of the operators. Parsing is then done by counting parentheses. Further steps construct a decision table.

111. **Sheridan**, P. B. The arithmetic translator-compiler of the IBM FORTRAN automatic coding system. *Commun. ACM*, 2:9–21, Feb. 1959. Amazingly compact description of a optimizing Fortran compiler; this digest covers only the translation of arithmetic expressions.
The expression is first turned into a fully parenthesized one, through a precedence-like scheme (**+** is turned into **))) + (((**, etc.). This leads to a list of triples (node number, operator, operand). This list is then reduced in several sweeps to eliminate copy operations and common subexpressions; these optimizations are machine-independent. Next several machine-dependent (for the IBM 704) optimizations are performed.

112. **Samelson**, K. and Bauer, F. L. Sequential formula translation. *Commun. ACM*, 3(2):76–83, Feb. 1960. (Parsing part only.) When translating a dyadic formula from left to right, the translation of an operator often has to be postponed because a later operator has a higher precedence. It is convenient to put such operators aside in a pushdown *cellar* (which later became known as a "stack"); the same applies to operands, for which an "address cellar" is introduced.
All parsing decisions can then be based on the most recent operator ξ in the cellar and the next input symbol α (sometimes called χ in the paper). If α is an operand, it is stacked on the address cellar and a new input symbol is read; otherwise a matrix is indexed with ξ and α, resulting in an action to be performed. This leads to a variant of operator precedence parsing.
The matrix (given in Table 1) was produced by hand from a non-existing grammar. It contains 5 different actions, two of which (1 and 3) are shifts (there is a separate shift to fill the empty stack). Action 5 is the general reduction for a dyadic operator, popping both the operator cellar and the address cellar. Action 4 handles one parentheses pair by discarding both ξ **(** **(** and α **(** **)**. Action 2 is a specialized dyadic reduction, which incorporates the subsequent shift; it is used when such a shift is guaranteed, as in two successive operators of the same precedence, and works by overwriting the top element in the cellar.

113. **Floyd**, Robert W. A descriptive language for symbol manipulation. *J. ACM*, 8:579–584, Oct. 1961. Original paper describing Floyd productions. See Section 9.3.2.

114. **Paul**, M. A general processor for certain formal languages. In *Symposium on Symbolic Languages in Data Processing*, pages 65–74, New York, 1962. Gordon and Breach. Early paper about the BRC(2,1) parser explained further in Eickel et al. [115]. Gives precise criteria under which the BRC(2,1) parser is deterministic, without explaining the parser itself.

115. **Eickel**, J., Paul, M., Bauer, F. L., and Samelson, K. A syntax-controlled generator of formal language processors. *Commun. ACM*, 6(8):451–455, Aug. 1963. In this paper, the authors develop and describe the BRC(2,1) parser already introduced by Paul [114]. The reduction rules in the grammar must have the form $U \leftarrow V$ or $R \leftarrow ST$. A set of 5 intuitively reasonable parse table construction rules are given, which assign to each combination $X_{n-1}X_n, t_k$ one of the actions $U \leftarrow X_n$, $R \leftarrow X_{n-1}X_n$, *shift* or *report error*. Here X_n is the top element of the stack and X_{n-1} the one just below it; t_k is the next input token.
An example of such a parse table construction rule is: if X_n can be reduced to a U such that $X_{n-1}U$ can be reduced to an R such that R can be followed by token t_k, then the table entry for $X_{n-1}X_n, t_k$, should contain $U \leftarrow \cdots \leftarrow X_n$. Note that chains of unit reductions are performed in one operation. The table is required to have no multiple entries. The terminology in the paper differs considerably from today's.

116. **Floyd**, Robert W. Syntactic analysis and operator precedence. *J. ACM*, 10(3):316–333, July 1963. Operator-precedence explained and applied to an ALGOL 60 compiler.

117. **Floyd**, Robert W. Bounded context syntax analysis. *Commun. ACM*, 7(2):62–67, Feb. 1964. For each right-hand side of a rule $A \rightarrow \alpha$ in the grammar, enough left and/or right context

is constructed (by hand) so that when α is found obeying that context in a sentential form in a left-to-right scan in a bottom-up parser, it can safely be assumed to be the handle. If you succeed, the grammar is *bounded-context*; if in addition the right hand contexts do not contain non-terminals, the grammar is *bounded-right-context*; analogously for bounded-left-context. A detailed example is given; it is BRC(2,1). The paper ends with a report of the discussion that ensued after the presentation of the paper.

118. **Wirth**, Niklaus and Weber, Helmut. EULER: A generalization of ALGOL and its formal definition, Part 1/2. *Commun. ACM*, 9(1/2):13–25/89–99, Jan. 1966. Detailed description of simple and extended precedence. A table generation algorithm is given. Part 2 contains the complete precedence table plus functions for the language EULER.

119. **Colmerauer**, A. *Précedence, analyse syntactique et languages de programmation*. PhD thesis, Technical report, Université de Grenoble, Grenoble, 1967, (in French). Defines two precedence schemes: total precedence, which is non-canonical, and left-to-right precedence, which is like normal precedence, except that some non-terminals are treated as if they were terminals. Some other variants are also covered, and an inclusion graph of the language types they define is shown, which includes some terra incognita.

120. **Bell**, James R. A new method for determining linear precedence functions for precedence grammars. *Commun. ACM*, 12(10):567–569, Oct. 1969. The precedence relations are used to set up a connectivity matrix. Take the transitive closure and count 1s in each row. Check for correctness of the result.

121. **Ichbiah**, J. and Morse, S. A technique for generating almost optimal Floyd-Evans productions of precedence grammars. *Commun. ACM*, 13(8):501–508, Aug. 1970. The notion of "weak precedence" is defined in the introduction. The body of the article is concerned with efficiently producing good Floyd-Evans productions from a given weak precedence grammar. The algorithm leads to production set sizes that are within 20% of the theoretical minimum.

122. **Loeckx**, Jacques. An algorithm for the construction of bounded-context parsers. *Commun. ACM*, 13(5):297–307, May 1970. The algorithm systematically generates all bounded-right-context (BRC) states the parser may encounter. Since BRCness is undecidable, the parser generator loops if the grammar is not BRC(m,n) for any value of m and n.

123. **McKeeman**, William M., Horning, James J., and Wortman, David B. *A Compiler Generator*. Prentice Hall, Englewood Cliffs, N.J., 1970. Good explanation of precedence and mixed-strategy parsing. Full application to the XPL compiler.

124. **Gray**, James N. and Harrison, Michael A. Canonical precedence schemes. *J. ACM*, 20(2):214–234, April 1973. The theory behind precedence parsing, unifying the schemes of Floyd [116], Wirth and Weber [118], and the canonical parser from Colmerauer [190]. Basically extends simple precedence by appointing some non-terminals as honorary terminals, the *strong operator set*; different strong operator sets lead to different parsers, and even to relationships with LR(k). Lots of math, lots of information. The paper emphasizes the importance of parse tree nodes being created in a clear and predictable order, in short "canonical".

125. **Levy**, M. R. Complete operator precedence. *Inform. Process. Lett.*, 4(2):38–40, Nov. 1975. Establishes conditions under which operator-precedence works properly.

126. **Henderson**, D. S. and Levy, M. R. An extended operator precedence parsing algorithm. *Computer J.*, 19(3):229–233, 1976. The relation $\lessdot$ is split into $\lessdot_1$ and $\lessdot_2$. $a \lessdot_1 b$ means that a may occur next to b, $a \lessdot_2 b$ means that a non-terminal has to occur between them. Likewise for $\doteq$ and $\gtrdot$. This is *extended operator-precedence*.

127. **Bertsch**, Eberhard. The storage requirement in precedence parsing. *Commun. ACM*, 20(3):192–194, March 1977. Suppose for a given grammar there exists a precedence matrix but the precedence functions f and g do not exist. There always exist sets of precedence functions f_i and g_j such that for two symbols a and b, comparison of $f_{c(b)}(a)$ and $g_{d(a)}(b)$ yields the precedence

relation between a and b, where c and d are selection functions which select the f_i and g_j to be compared. An algorithm is given to construct such a system of functions.

128. **Williams**, M. H. Complete operator precedence conditions. *Inform. Process. Lett.*, 6(2):60–62, April 1977. Revision of the criteria of Levy [125].

129. **Williams**, M. H. Conditions for extended operator precedence parsing. *Computer J.*, 22(2):164–168, 1979. Tighter analysis of extended operator-precedence than Henderson and Levy [126].

130. **Gonser**, Peter. *Behandlung syntaktischer Fehler unter Verwendung kurzer, fehlereinschließender Intervalle*. PhD thesis, Technical report, Technische Universität München, München, July 21 1981, (in German). The author's investigations on error treatment (see [308]) show that the precedence parsing algorithm has good error reporting properties because it allows the interval of the error to be securely determined. Since the existing precedence techniques are too weak, several new precedence grammars are proposed, often using existing terms and symbols ($\lesssim$, etc.) with new meanings.
1. An operator precedence grammar in which, for example, $a \lessdot b$ means that b can be the beginning of a non-terminal that can follow a, and $a \lesssim b$ means that b can be the first terminal in a right-hand side of a non-terminal that can follow a.
2. An extended operator precedence grammar in which two stack symbols, which must be a terminals, and a non-terminal form a precedence relation with the next input token.
3. An indexed operator precedence grammar, a grammar in which all terminal symbols are different. This virtually assures all kinds of good precedence properties; but hardly any grammar is indexed-operator. Starting from an LR(0) grammar it is, however, possible to construct a parsing algorithm that can disambiguate tokens on the fly during parsing, just in time for the precedence algorithm, by attaching LR state numbers to them. This distinguishes for example the `(` in function call `f(3)` from the `(` in the expression `x×(y+z)`. The proof of this theorem takes 19 pages; the algorithm itself another 5.
Each of these techniques comes with a set of rules for error correction.

131. **Williams**, M. H. A systematic test for extended operator precedence. *Inform. Process. Lett.*, 13(4-5):187–190, 1981. The criteria of Williams [129] in algorithmic form.

132. **Peyton Jones**, Simon L. Parsing distfix operators. *Commun. ACM*, 29(2):118–122, Feb. 1986. A distfix operator is an operator which is distributed over its operands; examples are `if . then . else . fi` and `rewrite . as . using . end`. It is useful to allow users to declare such operators, especially in functional languages.
Such distfix operators are introduced in a functional language using two devices. First the keywords of a distfix operator are given different representations depending on their positions: prefix keywords are written with a trailing dot, infix ones with a leading and a trailing dot, and postfix ones with a leading dot; so the user is required to write `rewrite. x .as. y .using. z .end`. These forms are recognized by the lexical analyzer, and given the token classes `PRE.TOKEN`, `IN.TOKEN`, and `END.TOKEN`. Second, generic rules are written in *yacc* to parse such structures.

133. **Aasa**, Annika. Precedences in specifications and implementations of programming languages. *Theoret. Comput. Sci.*, 142(1):3–26, May 1995. Fairly complicated but clearly explained algorithm for parsing expressions containing infix, prefix, postfix and distfix operators with externally given precedences. Even finding a sensible definition of the "correct" parsing is already difficult with those possibilities.

18.1.7 Finite-State Automata

134. **Mealy**, George H. A method for synthesizing sequential circuits. *Bell System Technical J.*, 34(5):1045–1079, 1955. Very readable paper on "sequential circuits" aka finite-state automata, except that the automata are built from relays connected by wires. The circuits consist

of AND, OR, and NOT gates, and delay units; the latter allow delayed feedback of signals from somewhere in the circuit to somewhere else. Starting from the circuit diagram, a set of input, output, excitation and state variables are defined, where the excitation variables describe the input to the delay units and the states their output. The delay units provide the finite-state memory. Since only the output variables are observable in response to the inputs, this leads naturally to attaching semantics to the transitions rather than to the (unobservable) states.
The relationships between the variables are recorded in "truth tables". These are shown to be equivalent to Moore's sequential machines. Moore's minimization procedure is then transformed so as to be applicable to truth tables. These then lead to minimal-size sequential circuits.
The rest of the paper dwells on the difficulties of asynchronous circuits, in which unknown delays may cause race conditions. The truth table method is reasonably good at handling them.

135. **Kleene**, S. C. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, Princeton, N.J., 1956. Introduces the Kleene star, but its meaning differs from the present one. An *event* is a $k \times l$ matrix, defining the k stimuli to k neurons over a time span of length l; a stimulus has the value 0 or 1. Events can be concatenated by just writing them one after another: EF means first there was an event E and then an event F; the final event F is in the present, and the train can then be applied to the set of neurons. Events can be repeated: EF, EEF, $EEEEF$, ... E^nF. Increasing the n introduces more and more events E in a more and more remote past, and since we do not usually know exactly what happened a long time ago, we are interested in the set E^0F, E^1F, E^2F, E^3F, ... E^nF for $n \to \infty$. This set is written as $E*F$, with a binary operator $*$ (not raised), and means "An occurrence of F preceded by any number of Es". The unary raised star does not occur in the paper, so its origin must be elsewhere.

136. **Moore**, E. F. Gedanken-experiments on sequential machines. In *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 129–153. Princeton University Press, Princeton, NJ, 1956. A finite-state automaton is endowed with an output function, to allow experiments with the machine; the machine is considered a black box. The output at a given moment is equal to the state of the FSA at that moment. Many, sometimes philosophical, conclusions are drawn from this model, culminating in the theorem that there is a sequence of at most $n^{nm+2}p^n/n!$ input symbols that distinguishes an FSA with n states, m different input symbols, and p different output symbols from any other such FSA.

137. **McNaughton**, R. and Yamada, H. Regular expressions and state graphs for automata. *IRE Transactions Computers*, EC-9(1):39–47, March 1960. Sets of sequences of input-output transitions are described by regular expressions, which are like regular expressions in CS except that intersection and negation are allowed. The output is generated the moment the automaton enters a state. A subset-like algorithm for converting regular expressions without intersection, negation, and ε-rules into FSAs is rigorously derived. The trouble-makers are introduced by repeatedly converting the innermost one into a well-behaved regular expression, using one of three conversion theorems. Note that the authors use ϕ for the empty sequence (string) and Λ for the empty set of strings (language).

138. **Brzozowski**, J. A. Canonical regular expressions and minimal state graphs for definite events. In *Symp. on Math. Theory of Automata*, pages 529–561, Brooklyn. N.Y., 1963. Brooklyn Politechnic. Getting unique minimal regular expressions from FSAs is difficult. The author defines a *definite event* as a regular set described by an expression of the form $E|\Sigma^*F$, where E and F are finite sets of finite-length strings. Using Brzozowski derivatives, the author gives an algorithm that will construct a definite event expression for any FSA that allows it.

139. **Brzozowski**, Janusz A. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. The author starts from regular expressions over [0,1] that use concatenation and Kleene star only, and then adds union, intersection, complement and exclusive-or. Next the *derivative* $D_s(R)$ of the regular language R with respect to s is defined as anything that can follow a prefix s in a sequence in R. Many theorems about these derivatives are proved, for example: "A sequence s is in R if and only if $D_s(R) = \varepsilon$. More importantly, it is shown that there are only a finite number of

different derivatives of a given R; these correspond to the states in the DFA. This is exploited to construct that DFA for regular expressions featuring the extended set of operations. Many examples. For an application of Brzozowski derivatives to XML validation see Sperberg-McQueen [359].

140. **Thompson**, Ken. Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. The regular expression is turned into a transition diagram, which is then interpreted in parallel. Remarkably, each step generates (IBM 7094) machine code to execute the next step.

141. **Aho**, Alfred V. and Corasick, Margaret J. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975. A given string embedded in a longer text is found by a very efficient FS automaton derived from that string.

142. **Krzemień**, Roman and Łukasiewicz, Andrzej. Automatic generation of lexical analyzers in a compiler-compiler. *Inform. Process. Lett.*, 4(6):165–168, March 1976. A grammar is *quasi-regular* if it features left or right recursion only; such grammars generate regular languages. A straightforward bottom-up algorithm is given to identify all quasi-regular subgrammars in a CF grammar, thus identifying its "lexical part", the part that can be handled by a lexical analyser in a compiler.

143. **Boyer**, Robert S. and Moore, J. Strother. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977. We want to find a string S of length l in a text T and start by positioning $S[1]$ at $T[1]$. Now suppose that $T[l]$ does not occur in S; then we can shift S to $T[l+1]$ without missing a match, and thus increase the speed of the search process. This principle can be extended to blocks of more characters.

144. **Ostrand**, Thomas J., Paull, Marvin C., and Weyuker, Elaine J. Parsing regular grammars with finite lookahead. *Acta Inform.*, 16:125–138, 1981. Every regular (Type 3) language can be recognized by a finite-state automaton without look-ahead, but such a device is not sufficient to do parsing. For parsing, look-ahead is needed; if a regular grammar needs a look-ahead of k tokens, it is called *FL(k)*. FS grammars are either FL(k), FL(∞) or ambiguous; a decision algorithm is described, which also determines the value of k, if appropriate.
A simple parsing algorithm is a FS automaton governed by a look-up table for each state, mapping look-aheads to new states. A second algorithm avoids these large tables by constructing the relevant look-ahead sets on the fly.

145. **Karp**, Richard M. and Rabin, Michael O. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development*, 31(2):249–260, 1987. We want to find a string S of length l in a text T. First we choose a hash function H that assigns a large integer to any string of length l and compute $H(S)$ and $H(T[1\cdots l])$. If they are equal, we compare S and $T[1\cdots l]$. If either fails we compute $H(T[2\cdots l+1])$ and repeat the process. The trick is to choose H so that $H(T[p+1\cdots p+l])$ can be computed cheaply from $H(T[p\cdots p+l-1])$. Note that this is not a FS algorithm but achieves a similar result.

146. **Jones**, Douglas W. How (not) to code a finite-state machine. *ACM SIGPLAN Notices*, 23(8):19–22, Aug. 1988. Small, well-structured and efficient code can be generated for a FS machine by deriving a single deterministic regular expression from the FS machine and implementing this expression directly using **while** and **repeat** constructions.

147. **Aho**, A. V. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science - Algorithms and Complexity, Vol. A*, pages 255–300. Elsevier, Amsterdam, The Netherlands, 1990. Chapter 5 of the handbook. Encyclopedic article on the subject, covering the state of the art in:
    single string matching:
        brute-force
        Karp-Rabin, caterpillar hash function
        Knuth-Morris-Pratt, automaton, forward
        Boyer-Moore, backward
    multiple string matching:

Aho-Corasick
Commentz-Walter, best description around
regular expression matching:
Thompson NFSA construction
regular expression matching with variables:
proved NP-complete
longest common substring location:
longest path in matrix
McCreight suffix trees
giving a very readable account of each of them, often with proof and complexity analysis. Draws amazing conclusions from Cook's Theorem: "Every 2-way deterministic pushdown automaton (2DPDA) language can be recognized in linear time on a random-access machine".
The paper ends with 139 literature references.

148. **Roche**, Emmanuel. Factorization of finite-state transducers. Technical Report TR-95-2, Mitsubishi Electric Research Laboratories, Cambridge, MA., Feb. 1995. A non-deterministic FSA F is decomposed into two deterministic ones by constructing a new graph on the states of F, in which arcs are present between each pair of states that can be reached by the same input string. This graph is then colored and the colors are considered new states. Two new automata are constructed, one which leads from the states of F to colors and one which leads from colors to states of F; they are constructed in such a way that they are deterministic. The concatenation C of these automata is equivalent to F. Often C is smaller than the traditional minimized deterministic equivalent of F, but of course it takes twice the time to do a transition.

149. **Watson**, Bruce W. A new regular grammar pattern matching algorithm. In Josep Díaz and Maria Serna, editors, *Algorithms: ESA '96, Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 364–377, Barcelona, Spain, Sept. 1996. Springer. Careful derivation of an algorithm, which applies the Boyer-Moore token-skipping technique [143] to regular expression matching.

150. **Brüggemann-Klein**, Anne and Wood, Derick. The validation of SGML content models. *Math. Comp. Modelling*, 25(4):73–84, 1997. The checking of an SGML file requires the construction of a FS automaton based on the document grammar. The paper gives criteria such that the automaton can be constructed in linear time.

151. **Laurikari**, Ville. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, Helsinki, Nov. 2001. Gives a linear-time algorithm for unambiguous substring parsing with a regular grammar, i.e., the algorithm returns a structured match for a regular expression matching a segment of the input. Unambiguity is enforced by three rules: longest possible match; longest possible subexpression match; and last possible match, in this order. Each transition in the NFA is augmented with a "tag", a variable which is set to the current input position when the transition is taken. A series of increasingly efficient but complicated algorithms for simulating tagged NFAs is given. Next it is shown how the gathered information can be used for creating a parse tree or to do approximate regular expression matching.
Chapters 4 and 5 report on the conversion of the tagged NFA to a tagged DFA, and on speed and memory usage tests, in which the tagged DFA performs between reasonably and spectacularly well. Excellent description and analysis of previous papers on finite-state parsing.

18.1.8 General Books and Papers on Parsing

152. **Aho**, Alfred V. and Ullman, Jeffrey D. *The Theory of Parsing, Translation and Compiling: Volume I: Parsing*. Prentice Hall, Englewood Cliffs, N.J., 1972. The book describes the parts of formal languages and automata theory relevant to parsing in a strict mathematical fashion. Since a considerable part of the pertinent theory of parsing had already been developed in 1972, the book is still reasonably up to date and is a veritable trove of definitions, theorems, lemmata and

proofs.
The required mathematical apparatus is first introduced, followed by a survey of compiler construction and by properties of formal languages. The rest of the book confines itself to CF and regular languages.
General parsing methods are treated in full: backtracking top-down and bottom-up, CYK and Earley. Directional non-backtracking methods are explained in detail, including general LL(k), LC(k) and LR(k), precedence parsing and various other approaches. A last chapter treats several non-grammatical methods for language specification and parsing.
Many practical matters concerning parser construction are treated in volume II, where the theoretical aspects of practical parser construction are covered; recursive descent is not mentioned, though.

153. **Backhouse**, Roland C. *Syntax of Programming Languages*. Prentice Hall, London, 1979. Grammars are considered in depth, as far as they are relevant to programming languages. FS automata and the parsing techniques LL and LR are treated in detail, and supported by lots of well-explained math. Often complete and efficient algorithms are given in Pascal. Much attention is paid to error recovery and repair, especially to least-cost repairs and locally optimal repairs. Definitely recommended for further reading.

154. **Nijholt**, Anton. Parsing strategies: A concise survey. In J. Gruska and M. Chytil, editors, *Mathematical Foundations of Computer Science*, volume 118 of *Lecture Notes in Computer Science*, pages 103–120. Springer-Verlag, Berlin, 1981. The context-free parser and language field is surveyed in terse prose. Highly informative to the connoisseur.

155. **Leermakers**, R. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, 1993. Parsing for the mathematically inclined, based on a formalism of the author's own creation. In fact the author proposes what seems to be a calculus for parsers: basic parsing problems are cast in the formalism, computations are performed on these formulas, and we arrive at new formulas that translate back into actual parsers, for example Earley or recursive ascent LR. These parsers have the form of functional programs.
The book contains a two-chapter introduction to the formalism, followed by chapters on applications to recursive descent, recursive ascent, parse forests, LR parsing, grammar transformations and attribute grammars. Some philosophical notes on these and other subjects end the book. The text is written in a deceptively simple but very clear prose, interleaved with considerable stretches of formulas.
The formalism has a high threshold, and requires considerable mathematical sophistication (Lambek types, etc.); but it has the clear and redeeming advantage that it is functional (excuse the pun): it allows actual computations to be performed and is not just an exposition aid.
For a review, see Schabes [157]. For an implementation see Sperber and Thiemann [356].

156. **Nederhof**, M.-J. *Linguistic Parsing and Program Transformation*. PhD thesis, Technical report, Katholieke Universiteit Nijmegen, Nijmegen, 1994. Contains in coherent chapter form versions of the following papers: "Generalized left-corner parsing" [172], "An Optimal Tabular Parsing Algorithm" [33], "Increasing the Applicability of LR Parsing" [94], and "Top-Down Parsing for Left-Recursive Grammars" [105], preceded by an introduction to parsing, and followed by a chapter on attribute propagation, and one on a grammar workbench.

157. **Schabes**, Yves. The functional treatment of parsing: Book review. *Computational Linguistics*, 21(1):112–115, 1995. Review of Leermakers [155]. Praises the approach and the courage. Criticizes the unusual formalism and some of the complexity analysis.

158. **Sikkel**, K. *Parsing Schemata*. Springer Verlag, 1996. Describes the primordial soup algorithm: the soup initially contains all grammar rules and all rules of the form $A \rightarrow t_i$ for all t_i in the input; both are in effect parse tree fragments. During stewing fragments are combined according to obvious rules, until all possible combinations have been formed. Then the complete parse trees float to the surface.
The rules of this algorithm, which is actually a transitive closure algorithm, are then formalized into sets of inference rules geared to parsing, called parsing schemata. These are then specialized to form many existing parsing methods and some new ones, including predictive head-corner parsing

[204]) and a parallel bottom-up GLR parser [233]. All this is supported by great mathematical rigor, but enough diagrams and examples are given to keep it readable.

18.2 Advanced Parsing Subjects

18.2.1 Generalized Deterministic Parsing

159. **Lang**, Bernard. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, Berlin, 1974. Explores the theoretical properties of doing breadth-first search to resolve the non-determinism in a bottom-up automaton with conflicts. See Tomita [160, 161, 162] for a practical realization.

160. **Tomita**, Masaru. LR parsers for natural languages. In *10th International Conference on Computational Linguistics*, pages 354–357. ACL, 1984. Two detailed examples of GLR parsing, on two English sentences. The parser features equal state combination, but no equal stack combination.

161. **Tomita**, Masaru. An efficient context-free parsing algorithm for natural languages. In *International Joint Conference on Artificial Intelligence*, pages 756–764, 1985. Explains GLR parsing in three steps: using stack lists, in which each concurrent LR parser has its own private stack; using tree-structured stacks, in which equal top states are combined yielding a forest of trees; and using the full graph-structured stacks. Also points out the defect in Earley's parse forest representation (Earley [14]), and shows that repairing it causes the algorithm to require more than $O(n^3)$ space on highly ambiguous grammars.

162. **Tomita**, Masaru. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Boston, 1986. Tomita describes an efficient parsing algorithm to be used in a "natural-language setting": input strings of some tens of words and considerable but not pathological ambiguity. The algorithm is essentially LR, starting parallel parses when an ambiguity is found in the LR-table. Full examples are given of handling ambiguities, lexical elements with multiple meanings and unknown lexical elements.
The algorithm is compared extensively to Earley's algorithm by measurement and it is found to be consistently five to ten times faster than the latter, in the domain for which it is intended. Earley's algorithm is better in pathological cases; Tomita's fails on unbounded ambiguity. No time bounds are given explicitly, but graphs show a behavior better than $O(n^3)$. Bouckaert, Pirotte and Snelling's algorithm [17]) is shown to be between Earley's and Tomita's in speed.
MacLisp programs of the various algorithms are given and the application in the Nishida and Doshita Machine Translation System is described.
For a review see Bańko [163].

163. **Bańko**, Mirosław. Efficient parsing for natural language: Book review. *Computational Linguistics*, 14(2):80–81, 1988. Two-column summary of Tomita's book [162].

164. **Billot**, Sylvie and Lang, Bernard. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151, June 1989. A parse forest resulting from parsing can be represented very conveniently by a grammar; subtrees are shared because they are represented by the same non-terminal. If the grammar is in 2-form (Sheil [20]), its size is $O(n^3)$, which is satisfactory.
To investigate this representation with various parsing schemes, the PDT interpreter of Lang [210] is implemented and LR(0), LR(1), LALR(1), LALR(2), weak precedence, and LL(0) transducers for some simple grammars are compared using it. A general observation is that parsers with great resolution power perform worse than weak precedence, because the overspecificness of the context prevents useful sharing of subtrees.

165. **Kipps**, James R. GLR parsing in time $O(n^3)$. In M. Tomita, editor, *Generalized LR Parsing*, pages 43–59. Kluwer Academic Publishers, Boston, 1991. Proves that the original GLR algorithm costs $O(n^{k+1})$ for grammars with rules of maximum length k. Identifies as cause of this complexity the searching of the graph-structured stack(GSS) during reduces. This process can take $O(i^l)$ actions at position i for a reduce of length l; worst case it has to be done for each input position, hence the $O(n^{k+1})$. The paper describes a memoization technique that stores for each node in the GSS and each distance $1 \leq p \leq k$ all nodes at distance p in an ancestors table ; this makes reduction $O(1)$, and when done cleverly the ancestors table can fully replace the GSS. Building the ancestors table costs $O(i^2)$ regardless of the grammar, hence the overall $O(n^3)$. For almost all grammars, however, the original algorithm is faster.
Contains explicit code for the original and the improved GLR algorithm.

166. **Lankhorst**, Marc. An empirical comparison of generalized LR tables. In R. Heemels, A. Nijholt, and K. Sikkel, editors, *Tomita's Algorithm: Extensions and Applications (TWLT1)*, number 91-68 in Memoranda Informatica in Twente Workshops on Language Technology, pages 87–93, Enschede, the Netherlands, 1991. Unversity of Twente. Lots of bar graphs, showing that as far as speed is concerned, LALR(1) wins by perhaps 5-10% over LR(0) and SLR(1), but that LR(1) is definitely worse. The reason is the large number of states, which reduces the number of common stack suffixes to be combined. In the end, the much simpler LR(0) is only a few percent sub-optimal.

167. **Nozohoor-Farshi**, R. GLR parsing for ε-grammars. In M. Tomita, editor, *Generalized LR Parsing*, pages 61–75. Kluwer Academic Publishers, Boston, 1991. Shows that Tomita's algorithm [162] loops on grammars with hidden left recursion where the left recursion can be hidden by unbounded many εs. Remedies this by constructing and pushing on the stack an FSA representing the unbounded string of εs, with its proper syntactic structure. This also happens to make the parser impervious to loops in the grammar, thus achieving full coverage of the CF grammars.

168. **Piastra**, Marco and Bolognesi, Roberto. An efficient context-free parsing algorithm with semantic actions. In *Trends in Artificial Intelligence*, volume 549 of *Lecture Notes in Artificial Intelligence*, pages 271–280. Springer-Verlag, Oct. 1991. A simple condition is imposed on the unit and ε-rules of a grammar, which controls the reductions in a reduce/reduce conflict in a GLR parser. The result is that the reductions can be done so that multiple values result for locally ambiguous segments of the input, and common stack suffixes can still be combined as usual.

169. **Rekers**, J. Generalized LR parsing for general context-free grammars. Technical Report CS-R9153, CWI, Amsterdam, 1991. Extensive pictorial explanation of the GLR algorithm, including parse forest construction, with full algorithms in a clear pseudo-code. The GLR parser in compiled LeLisp is 3 times slower than *yacc* on Pascal programs; the Earley parser drowned in garbage collection. On the other hand, Earley wins over GLR on highly ambiguous grammars.

170. **Deudekom**, A. van and Kooiman, P. Top-down non-correcting error recovery in LLgen. Technical Report IR 338, Vrije Universiteit, Faculteit Wiskunde en Informatica, Amsterdam, Oct. 1993. Describes the implementation of a Richter-style [313] error recovery mechanism in *LLgen*, an LL(1) parser generator, using a Generalized LL parser. The parser uses a reversed tree with loops as the data structure to store the predictions.
The error-recovering parser is an add-on feature and is activated only when an error has been found. It has to work with a grammar for suffixes of the original language, for which the LL(1) parser generator has no parse tables. So the parser uses the FIRST and FOLLOW sets only. Full algorithms are described.
A specialized garbage collector for the particular data structure was designed by Wattel and is described in the report. Its activation costs about 10% computing time, but saves large amounts of memory. Efficiency measurements are provided.
See [320] for the error-handling part.

171. **Merrill**, G. H. Parsing non-LR(k) grammars with yacc. *Softw. Pract. Exper.*, 23(8):829–850, 1993. This is generalized LR by *depth-first* rather than breadth-first search. LR conflicts in the Berkeley LALR(1) parser *byacc* are solved by recursively starting a subparser for each possibility. These parsers run in "trial mode", which means that all semantic actions except those specifically marked for trial are suppressed. Once the right path has been found, normal parsing continues along it. The design process and the required modifications to *byacc*, the lexical analyser, and the input grammar are described in detail.

172. **Nederhof**, Mark-Jan. Generalized left-corner parsing. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, pages 305–314, April 1993. A non-deterministic LC parser is extended to generalized parsing. This requires three problems to be solved to avoid non-termination: cycles, hidden left recursion, and ε-subtrees, subtrees that just produce ε. The hidden left recursion problem is solved by performing LC actions for any rule $A \rightarrow \mu B\beta$ when $\mu \stackrel{*}{\rightarrow} \varepsilon$; cycles are handled by creating loops in the parse tree under construction; and all empty subtrees are computed in advance. A special packing of the parse forest brings down the time and space complexity from $O(n^{p+1})$ where p is the length of the longest RHS to $O(n^3)$. Note that this technique is unrelated to the Generalized Left-Corner Parsing of Demers [103]. See also Chapter 2 of Nederhof's thesis [156].

173. **Lavie**, Aaron and Tomita, Masaru. GLR*: An efficient noise-skipping parsing algorithm for context-free grammars. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*, pages 183–200. Kluwer Academic Publishers, Dordrecht, 1996. The GLR* parser finds the longest subsequence of the input that is in the language; it does supersequence parsing. At each input token shifts are performed from all states that allow it; this implements skipping arbitrary segments of the input. A grading function is then used to weed out unwanted parsings. The algorithm has exponential complexity; to counteract this, the number of skipping shifts per token can be limited; a limit of 5 to 10 gives good results.

174. **Nederhof**, M.-J. and Satta, G. Efficient tabular LR parsing. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 239–246. Association for Computational Linguistics, 1996. Replaces the graph-structured stack of GLR parsing by the triangular table of CYK parsing, thus gaining efficiency and simplicity. The algorithm requires the grammar to be in "binary" form, which is Chomsky Normal form plus ε-rules. Explains how the very simple PDA used, among others, by Lang [210] can be obtained from the LR(0) table.

175. **Alonso Pardo**, M. A., Cabrero Souto, D., and Vilares Ferro, M. Construction of efficient generalized LR parsers. In Derick Wood and Sheng Yu, editors, *Second International Workshop on Implementing Automata*, volume 1436 of *Lecture Notes in Computer Science*, pages 7–24, Berlin, 1998. Springer-Verlag. Systematic derivation of an $O(n^3)$ GLR parsing algorithm from the Earley parser. First the Earley parser is rewritten as a dynamic programming algorithm. Next the Earley sets are compiled into sets of LR(0) states. Then look-ahead is introduced leading to LR(1) states, which are then combined into LALR(1) states. And finally implicit binarization is used to achieve the $O(n^3)$ complexity. The resulting parser consists of a considerable number of set definitions. It is about 5 times faster than the GLR parser from Rekers [169].

176. **Aycock**, John and Horspool, R. Nigel. Faster generalized LR parsing. In *Compiler Construction: 8th International Conference, CC'99*, volume 1575 of *Lecture Notes in Computer Science*, pages 32–46, Berlin, 1999. Springer Verlag. The stack is needed only for non-left recursion in an LR parser; everything else can be done by a DFA on the top of the stack. Recursion points (called "limit points" in the paper) are identified in the grammar using a heuristic form of the feedback arc set (FAS) algorithm. The grammar is broken at those points; this yields a regular grammar for which a DFA is constructed. Only when the DFA reaches a limit point, stack actions are initiated. The resulting very fast LR parser is used as a basis for a GLR parser. See also Aycock et al. [178].

177. **Scott**, Elizabeth, Johnstone, Adrian, and Hussain, Shamsa Sadaf. Tomita-style generalised LR parsers. Technical report, Royal Holloway, University of London, London, Dec. 2000. GLR parsers are bothered by nullable non-terminals at the beginning and end of a rule; those at the beginning cause errors when they hide left recursion; those at the end cause gross inefficiencies. The Generalized Reduction-Modified LR parser GRMLR solves the first problem by using an improved version of Nozohoor-Farshi's solution [167]. It solves the second problem by using an item $A \rightarrow \alpha\bullet\beta$ as a reduce item when β is nullable; a rule $A \rightarrow \alpha\beta$ with β nullable is called *right-nullable*.
For grammars that exhibit these problems a gain of roughly 30% is obtained. Full algorithm and correctness proof given.

178. **Aycock**, John, Horspool, R. Nigel, Janoušek, Jan, and Melichar, Bořivoj. Even faster generalized LR parsing. *Acta Inform.*, 37(9):633–651, 2001. Actions on the graph-structured stack are the most expensive items in generalized LR parsing and the fewer are required the better. For grammars without right recursion or hidden left recursion the stack actions between two shifts can be combined into two batches, a pop sequence and a push sequence. The optimization saves between 70% (for an unambiguous grammar) and 90% (for a highly ambiguous grammar) on processing time.

179. **Fortes Gálvez**, José, Farré, Jacques, and Aguiar, Miguel Ángel Pérez. Practical nondeterministic DR(k) parsing on graph-structured stack. In *Computational Linguistics and Intelligent Text Processing*, volume 2004 of *Lecture Notes in Computer Science*, pages 411–422. Springer Verlag, 2001. Generalized DR parsing. Applying the LR-to-DR table conversion of [95] does not work if the LR table has multiple entries, so a direct DR table construction algorithm is presented, which is capable of producing a non-deterministic DR table. A GSS algorithm using this table is described. Explicit algorithms are given.

180. **Johnstone**, Adrian and Scott, Elizabeth. Generalised reduction modified LR parsing for domain specific language prototyping. In *35th Hawaii International Conference on System Sciences*, page 282. IEEE, 2002. Summary of Scott et al. [177].

181. **Johnstone**, Adrian and Scott, Elizabeth. Generalised regular parsers. In *Compiler Construction: 12th International Conference, CC'03*, volume 2622 of *Lecture Notes in Computer Science*, pages 232–246. Springer Verlag, 2003. The grammar is decomposed into a regular grammar and a set of recursive grammars as follows. All derivations of the form $A \xrightarrow{*} \alpha A \beta$ with α and β not empty are blocked by replacing the A in the right-hand side of a rule involved in this derivation by a special symbol $A^{\perp}$. This yields the regular grammar; it is transformed into an NFA whose arcs are labeled with terminals, left- or right-recursive rule numbers $\mathcal{R}n$, or ε; this is a *Reduction Incorporated Automaton* (RIA). Next a *Recursive Call Automaton* (RCA) is constructed for each thus suppressed A. Each such automaton is then connected to the NFA by transitions marked with $push(A)$ and *pop*, in a way similar to that of ATNs. Finally the εs are removed using the subset algorithm; any other non-determinism remains. The resulting automaton is grafted on a graph-structured stack in GLR fashion. When the automaton meets a $push(A)$ transition, return info is stacked and the automaton proceeds to recognize an A; upon *pop* it returns.
The resulting parser operates with a minimum of stack operations, and with zero stack operations for almost all CF grammars that define a regular language. For proofs, etc. see Scott and Johnstone [183].

182. **Scott**, E., Johnstone, A., and Economopoulos, G. R. BRN-table based GLR parsers. Technical Report CSD-TR-03-06, CS Dept., Royal Holloway, University of London, London, July 2003. After a detailed informal and formal description of the GRMLR parser [177], called "RNGLR" for "right-nullable GLR" here, the notion "binary right-nullable", or BRN, is introduced, for the purpose of making the GLR parser run in $O(n^3)$ on all grammars. In BRN the LR(1) table is modified so that each reduction grabs at most 2 stack elements. This makes the GLR parser react as if the longest right-hand side is at most 2 long, and since GLR parsing is $O(n^{k+1})$, where k is the length of the longest right-hand side, $O(n^3)$ complexity results.

Many examples, many pictures, much explicit code, many proofs, extensive complexity results, many of them in closed formula forms, etc. With so many goodies it lacks an index.

183. **Scott**, Elizabeth and Johnstone, Adrian. Table based parsers with reduced stack activity. Technical Report CSD-TR-02-08, CS Dept., Royal Holloway, University of London, London, May 2003. Proofs, examples and background information for Johnstone and Scott [181].

184. **Johnstone**, Adrian, Scott, Elizabeth, and Economopoulos, Giorgios R. Generalised parsing: Some costs. In *Compiler Construction: 13th International Conf. CC'2004*, volume 2985 of *Lecture Notes in Computer Science*, pages 89–103, Berlin, 2004. Springer-Verlag. Several GLR techniques are compared experimentally and the effects found are discussed. The answers depend on many factors, including available memory size; for present-day grammars and machines RNGLR is a good choice.

185. **Johnstone**, Adrian and Scott, Elizabeth. Recursion engineering for reduction incorporated parsers. *Electr. Notes Theor. Comput. Sci.*, 141(4):143–160, 2005. Reduction-incorporated parsers require the grammar to be split in a regular part and a set of recursive non-terminals, where we want the regular part to be large and the recursive part to be small. We can make the regular part larger and larger by substituting out more and more non-terminals. The tables that correspond to optimum parsing speed can be enormous, and trade-offs have to be made. Heuristics, profiling, and manual intervention are considered, the latter based on the visualization tool *VCG*.

186. **Scott**, Elizabeth and Johnstone, Adrian. Generalised bottom up parsers with reduced stack activity. *Computer J.*, 48(5):565–587, 2005. The Reduction Incorporated (RI) technique from Johnstone and Scott [181] and Scott and Johnstone [183] is incorporated in a table-driven bottom-up parser, yielding a "shared packed parse forest" (SPPF). Run-time data structures can be an order of magnitude or more smaller than those of a GSS implementation. Extensive implementation code, proofs of correctness, efficiency analyses.

187. **Johnstone**, Adrian, Scott, Elizabeth, and Economopoulos, Giorgios R. Evaluating GLR parsing algorithms. *Sci. Comput. Progr.*, 61(3):228–244, 2006. A clear exposition of two improvements of Nozohoor-Farshi's modification [167] to Tomita's algorithm, the Right Nulled GLR (RNGLR) algorithm [182], and the Binary Right Nulled GLR (BRNGLR) algorithm [182] is followed by an extensive comparison of these methods, using LR(0), SLR(1) and LR(1) tables for grammars for C, Pascal and Cobol. The conclusion is that Right Nulled GLR (RNGLR) with an SLR(1) table performs adequately except in bizarre cases.

188. **Scott**, Elizabeth and Johnstone, Adrian. Right nulled GLR parsers. *ACM Trans. Prog. Lang. Syst.*, 28(4):577–618, 2006. After a 9 page(!) history of parsing since the time that the parsing problem was considered solved (mid-1970s), the principles of GLR parsing and right-nulled LR(1) (RN) parsing (Scott [100, 177]) are explained and combined in the RNGLR algorithm. The resulting recognizer is then extended to produce parse trees. Depending on the nature of the grammar, using right-nulled LR(1) can help considerably: on one grammar RNGLR visits only 25% of the edges visited by the standard GLR algorithm. Extensive implementation code, proofs of correctness, efficiency analyses.

18.2.2 Non-Canonical Parsing

This section covers the bottom-up non-canonical methods; the top-down ones (LC, etc.) are collected in (Web)Section 18.1.5.

189. **Floyd**, Robert W. Bounded context syntax analysis. *Commun. ACM*, 7(2):62–67, Feb. 1964. For each right-hand side of a rule $A \rightarrow \alpha$ in the grammar, enough left and/or right context

is constructed (by hand) so that when α is found obeying that context in a sentential form in a left-to-right scan in a bottom-up parser, it can safely be assumed to be the handle. If you succeed, the grammar is *bounded-context*. A complicated set of rules is given to check if you have succeeded. See [117] for the bounded-right-context part.

190. **Colmerauer**, Alain. *Précedence, analyse syntactique et languages de programmation*. PhD thesis, Technical report, Université de Grenoble, Grenoble, 1967, (in French). Defines total precedence and left-to-right precedence. See [119].

191. **Colmerauer**, Alain. Total precedence relations. *J. ACM*, 17(1):14–30, Jan. 1970. The non-terminal resulting from a reduction is not put on the stack but pushed back into the input stream; this leaves room for more reductions on the stack. This causes precedence relations that differ considerably from simple precedence.

192. **Szymanski**, T. G. *Generalized Bottom-up Parsing*. PhD thesis, Technical Report TR73-168, Cornell University, Ithaca, N.Y., 1973. For convenience derivation trees are linearized by, for each node, writing down its linearized children followed by a token $\texttt{]}_{\texttt{n}}$, where $\texttt{n}$ is the number of the production rule. For a given grammar G all its sentential forms form a language in this notation: G's *description language*. Define a "phrase" as a node that has only leaves as children. Now suppose we delete from derivation trees all nodes that are not phrases, and linearize these. This results in the *phrase language* of G. The point is that phrases can be reduced immediately, and consequently the phrase language contains all possibilities for immediate reduces.
Phrase languages are a very general model for bottom-up parsing. Consider a phrase P in a phrase language. We can then compute the left and right contexts of P, which turn out to be CF languages. The construct consisting of the left context of P, P, and right context of P is a *parsing pattern* for P. A complete set of mutually exclusive parsing patterns G is a *parsing scheme* for G. It is undecidable if there is a parsing scheme for a given grammar.
The problem can be made manageable by putting restrictions on the parsing patterns. Known specializations are bounded-right-context (Floyd [117]), LR(k) (Knuth [52]), LR-regular (Čulik, II and Cohen [57]), and bounded-context parsable (Williams [193]). New specializations discussed in this thesis are *FPFAP(k)*, where regular left and right contexts are maintained and used in a left-to-right scan with a k-token look-ahead; LR(k,∞) and LR(k,t), in which the left context is restricted to that constructed by LR parsing; and *RPP*, *Regular Pattern Parsable*, which is basically FPFAP(∞).
The rest (two-thirds) of the thesis explores these new methods in detail. LR(k) and SLR(k) are derived as representations of inexact-context parsing. A section on the comparison of these methods as to grammars and languages and a section on open problems conclude the thesis.

193. **Williams**, John H. Bounded-context parsable grammars. *Inform. Control*, 28(4):314–334, Aug. 1975. The bounded-context parser without restrictions on left and right context, hinted at by Floyd [189], is worked out in detail; grammars allowing it are called bounded-context parsable, often abbreviated to BCP. All LR languages are BCP languages but not all LR grammars are BCP grammars. BCP grammars allow, among others, the parsing in linear time of some non-deterministic languages. Although a parser could be constructed, it would not be practical.

194. **Szymanski**, Thomas G. and Williams, John H. Non-canonical extensions of bottom-up parsing techniques. *SIAM J. Computing*, 5(2):231–250, June 1976. Theory of non-canonical versions of several bottom-up parsing techniques, with good informal introduction. Condensation of Szymanski's thesis [192].

195. **Friede**, Dietmar. Transition diagrams and strict deterministic grammars. In Klaus Weihrauch, editor, *4th GI-Conference*, volume 67 of *Lecture Notes in Computer Science*, pages 113–123, Berlin, 1978. Springer-Verlag. Explores the possibilities to parse strict deterministic grammars (a large subset of LR(0)) using transition diagrams, which are top-down. This leads to PLL(k) grammars, which are further described in Friede [196].

196. **Friede**, Dietmar. Partitioned LL(k) grammars. In H.A. Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 245–255. Springer-Verlag, Berlin, 1979. The left factorization, usually performed by hand, which

turns a rule like $A \rightarrow PQ|PR$ into $A \rightarrow PZ$; $Z \rightarrow Q|R$, is incorporated into the parsing algorithm in a very general and recursive way. This results in the PLL(k) grammars and their languages. The resulting grammars are more like LC and LR grammars than like LL grammars. Many theorems, some surprising, about these grammars and languages are proved; examples are: 1. the PLL(1) grammars include the LL(1) grammars. 2. the PLL(0) grammars are exactly the strict deterministic grammars. 3. the classes of PLL(k) languages are all equal for $k > 0$. 4. the PLL(0) languages form a proper subset of the PLL(1) languages. Theorems (2), (3) and (4) also hold for LR, but a PLL parser is much simpler to construct.

197. **Tai**, Kuo-Chung. Noncanonical SLR(1) grammars. *ACM Trans. Prog. Lang. Syst.*, 1(2):295–320, Oct. 1979. An attempt is made to solve reduce/reduce conflicts by postponing the decision, as follows. Suppose two reduce items $A \rightarrow \alpha\bullet$ and $B \rightarrow \beta\bullet$ with overlapping look-aheads in an item set I. The look-ahead for the A item is replaced by LM_FOLLOW(A), the set of non-terminals that can follow A in any *leftmost* derivation, same for the look-ahead of B, and all initial items for these non-terminals are added to I. Now I will continue to try to recognize the above non-terminals, which, once found can be used as look-ahead non-terminals to resolve the original reduce/reduce conflict. This leads to two non-canonical parsing methods LSLR(1) and NSLR(1), which differ in details.

198. **Proudian**, Derek and Pollard, Carl J. Parsing head-driven phrase structure grammar. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 167–171, 1985. The desirability of starting analysis with the "head" of a phrase is argued on linguistic grounds. Passing of features between parents and children is automatic, allowing a large part of English to be represented by 16 rules only. Parsing is chart parsing, in which the order in which edges are added is not left-to-right, but rather controlled by head information and the unification of features of children.

199. **Kay**, Martin. Head-driven parsing. In *International Workshop on Parsing Technologies*, pages 52–62, 1989. Since the complements of a non-terminal (= the structures it governs in linguistic terms) are often more important than textual adjacency, it is logical and profitable to parse first the section that supplies the most information. This is realized by appointing one of the members in each RHS as the "head". Parsing then starts by finding the head of the head etc. of the start symbol; usually it is a verb form which then gives information about its subject, object(s), etc. Finding the head is awkward, since it may be anywhere in the sentence. A non-directional chart parser is extended with three new types of arcs, pending, current and seek, which assist in the search. Also, a Prolog implementation of an Unger parser is given which works on a grammar in 2-form: if the head is in the first member of an alternative, searching starts from the left, otherwise from the right. The advantages of head-driven parsing are conceptual; the author expects no speed-up.

200. **Salomon**, Daniel J. and Cormack, Gordon V. Scannerless NSLR(1) parsing of programming languages. *ACM SIGPLAN Notices*, 24(7):170–178, July 1989. The traditional CF syntax is extended with two rule types: an exclusion rule $A \nrightarrow B$, which means that any sentential form in which A generates a terminal production of B (with B regular) is illegal; and an adjacency restriction $A \not\leftarrow B$ which means that any sentential form in which terminal productions of A and B are adjacent, is illegal. The authors show that the addition of these two types of rules allows one to incorporate the lexical phase of a compiler into the parser. The system uses a non-canonical SLR(1) parser.

201. **Satta**, Giorgio and Stock, Oliviero. Head-driven bidirectional parsing: A tabular method. In *International Workshop on Parsing Technologies*, pages 43–51, 1989. The Earley algorithm is adapted to head grammars, as follows. A second dot is placed in each Earley item for a section $w_{m,n}$ of the input, not coinciding with the first dot, with the meaning that the part between the dots produces $w_{m,n}$. Parsing no longer proceeds from left to right but according to an action pool, which is prefilled upon initialization, and which is processed until empty.
The initialization creates all items that describe terminal symbols in the input that are heads in any production rule. Processing takes one item from the action pool, and tries to perform five actions on it, in arbitrary order: extend the left dot in an uncompleted item to the left, likewise to the right,

use the result of the completed item to extend a left dot to the left, likewise to the right, and use a completed item to identify a new head in some production rule.
The presented algorithm contains an optimization to prevent subtrees to be recognized twice, by extending left then right, and by extending right then left.

202. **Hutton**, Michael D. Noncanonical extensions of LR parsing methods. Technical report, University of Waterloo, Waterloo, Aug. 1990. After a good survey of existing non-canonical methods, the author sets out to create a non-canonical LALR(k) (*NLALR(k)*) parser, analogous to Tai's NSLR(1) from SLR(1), but finds that it is undecidable if a grammar is NLALR(k). The problem is solved by restricting the number of postponements to a fixed number t, resulting in *NLALR(k,t)*, also called *LALR(k,t)*.

203. **Nederhof**, M.-J. and Satta, G. An extended theory of head-driven parsing. In *32nd Annual Meeting of the Association for Computational Linguistics*, pages 210–217, June 1994. The traditional Earley item $[A \to \alpha\bullet\beta, i]$ in column j is replaced by a position-independent double-dotted item $[i,k,A \to \alpha\bullet\gamma\bullet\beta, m, p]$ with the meaning that a parsing of the string $a_{i+1}\cdots a_p$ by $A \to \alpha\gamma\beta$ is sought, where γ already produces $a_{k+1}\cdots a_m$. This collapses into Earley by setting $\alpha = \varepsilon$, $k = i$, $p = n$ where n is the length of the input string, and putting the item in column m; the end of the string sought in Earley parsing is not known, so $p = n$ can be omitted.
Using these double-dotted items, Earley-like algorithms are produced basing the predictors on top-down parsing, head-corner parsing (derived from left-corner), predictive head-infix (HI) parsing (derived from predictive LR), extended HI parsing (derived from extended LR), and HI parsing (extended from LR), all in a very compact but still understandable style. Since head parsing is by nature partially bottom-up, ε-rules are a problem, and the presented algorithms do not allow them.
Next, head grammars are generalized by requiring that the left and right parts around the head again have sub-heads, and so on recursively. A parenthesized notation is given: $S \to ((c)A(b))s$, in which s is the head, A the sub-head of the left part, etc. The above parsing algorithm is extended to these generalized head grammars. Correctness proofs are sketched.

204. **Sikkel**, K. and Akker, R. op den. Predictive head-corner chart parsing. In Harry Bunt and Masaru Tomita, editors, *Recent Advances in Parsing Technology*, pages 113–132. Kluwer Academic Publishers, Dordrecht, 1996. Starting from the start symbol, the heads are followed down the grammar until a terminal t is reached; this results in a "head spine". This terminal is then looked up in the input, and head spines are constructed to each position p_i at which t occurs. Left and right arcs are then predicted from each spine to p_i, and the process is repeated recursively for the head of the left arc over the segment $1..p_i - 1$ and for the head of the right arc over the segment $p_i + 1..n$.

205. **Noord**, Gertjan van. An efficient implementation of the head-corner parser. *Computational Linguistics*, 23(3):425–456, 1997. Very carefully reasoned and detailed account of the construction of a head-corner parser in Prolog, ultimately intended for speech recognition. Shows data from real-world experiments. The author points out that memoization is efficient for large chunks of input only.

206. **Madhavan**, Maya, Shankar, Priti, Rai, Siddharta, and Ramakrishna, U. Extending Graham-Glanville techniques for optimal code generation. *ACM Trans. Prog. Lang. Syst.*, 22(6):973–1001, Nov. 2000. (Parsing part only.) Classical Graham–Glanville code generation is riddled by ambiguities that have to be resolved too early, resulting in sub-optimal code. This paper describes a parsing method which the authors do not seem to name, and which allows ambiguities in an LR-like parser to remain unresolved arbitrarily long.
The method is applicable to grammars that have the following property; no technical name for such grammars is given. All rules are either "unit rules" in which the right-hand side consists of exactly one non-terminal, or "operator rules" in which the right-hand side consists of N (≥ 0) non-terminals followed by a terminal, the "operator". As usual with operators, the operator has an arity, which has to be equal to N.
In such a grammar, each shift of a terminal is immediately followed by a reduce, and the arity of the terminal shifted determines the number of items on the stack that are replaced by one non-terminal.

This allows us to do the reduction, even if multiple reductions are possible, without keeping multiple stacks as is done in a GLR parser: all reduces take away the same number of stack items. Note that all the items on the stack are non-terminals. Such a reduction results in a set of non-terminals to be pushed on the stack, each with a different, possibly ambiguous parse tree attached to them. This set may then be extended by other non-terminals, introduced by unit reductions using unit rules; only when no further reduces are possible is the next terminal (= operator) shifted in.
A new automaton is constructed from the existing LR(0) automaton, based on the above parsing algorithm; the unit reductions have been incorporated completely into the automaton. The algorithm to do so is described extensively.
The parser is used to parse the intermediate code stream in a compiler and to isolate in it operator trees that correspond to machine instructions, the grammar rules. A cost is attached to each rule, and the costs are used to disambiguate the parse tree and so decide on the machine code to be generated.

207. **Farré**, Jacques and Fortes Gálvez, José. A basis for looping extensions to discriminating-reverse parsing. In *5th Internat. Conf. Implementation and Applications of Automata, CIAA 2000*, volume 2088 of *Lecture Notes in Computer Science*, pages 122–134, 2001. Since DR parsers require only a small top segment of the stack, they can easily build up enough left context after a DR conflict to do non-canonical DR (NDR). When a conflict occurs, a state-specific marker is pushed on the stack, and input symbols are shifted until enough context is assembled. Then DR parsing can resume normally on the segment above the marker. The shift strategy is guided by a mirror image of the original DR graph. This requires serious needlework to the graphs, but complete algorithms are given. This technique shows especially clearly that non-canonical parsing is actually doing a CF look-ahead.

208. **Farré**, Jacques and Fortes Gálvez, José. Bounded-graph construction for noncanonical discriminating-reverse parsers. In *6th Internat. Conf. Implementation and Applications of Automata, CIAA 2001*, volume 2494 of *Lecture Notes in Computer Science*, pages 101–114. Springer Verlag, 2002. Improvements to the graphs construction algorithm in [207].

209. **Farré**, Jacques and Fortes Gálvez, J. Bounded-connect noncanonical discriminating-reverse parsers. *Theoret. Comput. Sci.*, 313(1):73–91, 2004. Improvements to [208]. More theory of non-canonical DR parsers, defining BC(h)DR(0).

18.2.3 Substring Parsing

210. **Lang**, Bernard. Parsing incomplete sentences. In D. Vargha, editor, *12th International Conf. on Comput. Linguistics COLING'88*, pages 365–371. Association for Computational Linguistics, 1988. An incomplete sentence is a sentence containing one or more unknown symbols (represented by `?`) and/or unknown symbol sequences (represented by `*`). General left-to-right CF parsers can handle these inputs as follows. Upon seeing `?` make transitions on all possible input symbols while moving to the next position; upon seeing `*` make transitions on all possible input symbols while staying at the same position. The latter process requires transitive closure.
These features are incorporated into an all-paths non-deterministic interpreter of pushdown transducers. This PDT interpreter accepts transitions of the form $(p\ A\ a\ \rightarrow\ q\ B\ u)$, where p and q are states, A and B stack symbols, a is an input token, and u is an output token, usually a number of a production rule. A, B, a and/or u may be missing, and the input may contain wild cards. Note that these transitions can push and pop only one stack symbol at a time; transitions pushing or popping more than one symbol have to be decomposed. The interpretation is performed by constructing sets of Earley-like items between successive input tokens; these items then form the non-terminals of the output grammar. Given the form of the allowed transitions, the output grammar is automatically in 2-form, but may contain useless and unreachable non-terminals. The grammar produces the input string as many times as there are ambiguities, interlaced with output tokens which tell how the preceding symbols must be reduced, thus creating a genuine parse tree.

Note that the "variation of Earley's algorithm" from the paper is not closely related to Earley, but is rather a formalization of generalized LR parsing. Likewise, the items in the paper are only remotely related to Earley items. The above transitions on `?` and `*` are, however, applicable independent of this.

211. **Cormack**, Gordon V. An LR substring parser for noncorrecting syntax error recovery. *ACM SIGPLAN Notices*, 24(7):161–169, July 1989. The LR(1) parser generation method is modified to include suffix items of the form $A \rightarrow \cdots \bullet \beta$, which mean that there exists a production rule $A \rightarrow \alpha\beta$ in the grammar and that it can be the handle, provided we now recognize β. The parser generation starts from a state containing all possible suffix items, and proceeds in LR fashion from there, using fairly obvious shift and reduce rules. If this yields a deterministic parser, the grammar was BC-LR(1,1); it does so for any bounded-context(1,1) grammar, thereby confirming Richter's [313] conjecture that linear-time suffix parsers are possible for BC grammars. The resulting parser is about twice as large as an ordinary LR parser. A computationally simpler BC-SLR(1,1) variant is also explained. For the error recovery aspects see the same paper [318].

212. **Rekers**, Jan and Koorn, Wilco. Substring parsing for arbitrary context-free grammars. *ACM SIGPLAN Notices*, 26(5):59–66, May 1991. A GLR parser is modified to parse substrings, as follows. The parser is started in all LR states that result from shifting over the first input symbol. Shifts are handled as usual, and so are reduces that find all their children on the stack. A reduce to $A \rightarrow \alpha$, where A contains more symbols than the stack can provide, adds all states that can be reached by a shift over A. A technique is given to produce trees for the completion of the substring, to be used, for example, in an incremental editor.

213. **Rekers**, J. *Parser Generation for Interactive Environments*. PhD thesis, Technical report, Leiden University, Leiden, 1992. Same as [347]. Chapter 4 discusses the substring parser from [212].

214. **Bates**, Joseph and Lavie, Alon. Recognizing substrings of LR(k) languages in linear time. *ACM Trans. Prog. Lang. Syst.*, 16(3):1051–1077, 1994. Reporting on work done in the late 1970s, the authors show how a GLR parser can be modified to run in linear time when using a conflict-free LR table. Basically, the algorithm starts with a GLR stack configuration consisting of all possible states, and maintains as large a right hand chunk of the GLR stack configuration as possible. This results in a forest of GLR stack configurations, each with a different state at the root; each path from the root is a top segment of a possible LR stack, with the root as top of stack. For each token, a number of reduces is performed on all trees in the forest, followed by a shift, if possible. Then all trees with equal root states are merged. If a reduce $A \rightarrow \alpha$ reduces more stack than is available, new trees result, each consisting of a state that allows a shift on A. When two paths are merged, the shorter path wins, since the absence of the rest of a path implies all possible paths, which subsumes the longer path. Explicit algorithm and proofs are given. See Goeman [218] for an improved version.

215. **Bertsch**, Eberhard. An asymptotically optimal algorithm for non-correcting LL(1) error recovery. Technical Report 176, Ruhr-Universität Bochum, Bochun, Germany, April 1994. First a suffix grammar G_S is created from the LL(1) grammar G. Next G_S is turned into a left-regular grammar L by assuming its CF non-terminals to be terminals; this regular grammar generates the "root set" of G. Then a linear method is shown to fill in the recognition table in linear time, by doing tabular LL(1) parsing using grammar G. Now all recognizable non-terminals in the substring are effectively terminals, but of varying size. Next a second tabular parser is explained to parse the non-terminals according to the left-recursive grammar L; it is again linear. Finally the resulting suffix parser is used to do non-correcting error recovery.

216. **Nederhof**, Mark-Jan and Bertsch, Eberhard. Linear-time suffix parsing for deterministic languages. *J. ACM*, 43(3):524–554, May 1996. Shows that an Earley parser working with a conflict-free LL(1) parse table runs in linear time. Next extends this result to suffix parsing with an Earley parser. The LR case is more complicated. The language is assumed to be described by a very restricted pushdown automaton, rather than by a CF grammar. Using this automaton in suffix parsing with an Earley parser rather than an LL(1) parse table results in an $O(n^2)$ algorithm. To

avoid this the automaton is refined so it consumes a token on every move. The Earley suffix parser using this automaton is then proven to be linear. Several extensions and implementation ideas are discussed. See Section 12.3.3.2.

217. **Ruckert**, Martin. Generating efficient substring parsers for BRC grammars. Technical Report 98-105, State University of New York at New Paltz, New Paltz, NY 12561, July 1998. All BRC(m,n) parsing patterns are generated and subjected to Floyd's [117] tests; BRC(m,n) (bounded-right-context) is BCP(m,n) with the right context restricted to terminals. If a complete set remains, then for every correct sentential form there is at least one pattern which identifies a handle; this handle is not necessarily the leftmost one, so the parser is non-canonical – but it is linear. Since this setup can start parsing anew wherever it wants to, it identifies correct substrings in a natural way, if the sentential form is not correct. Heuristics are given to improve the set of parsing patterns. The paper is written in a context of error recovery.

218. **Goeman**, Heiko. On parsing and condensing substrings of LR languages in linear time. *Theoret. Comput. Sci.*, 267(1-2):61–82, 2001. Tidied-up version of Bates and Lavie's algorithm [214], with better code and better proofs. The algorithm is extended with memoization, which condenses the input string as it is being parsed, thus increasing reparsing speed.

18.2.4 Parsing as Intersection

219. **Bar-Hillel**, Y., Perles, M., and Shamir, E. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961. The intersection of a CF grammar and a FS automaton is constructed in a time $O(n^d+1)$, where n is the number of states in the automaton, and d is the maximum length of the RHSs in the grammar. For more aspects of the paper see [386].

220. **Lang**, Bernard. A generative view of ill-formed input processing. In *ATR Symposium on Basic Research for Telephone Interpretation*, Dec. 1989. Proposes weighted grammars (à la Lyon [294]) for the treatment of various ill or problematically formed input, among which word lattices. A word lattice is a restricted form of FSA, but even general FSAs may appear as input when sequences of words are missing or partially identified. The author notes in passing that "the parsing of an FSA A according to a CF grammar G can produce a new grammar T for the intersection of the languages $\mathcal{L}(A)$ and $\mathcal{L}(G)$, giving to all sentences in that intersection the same structure as the original grammar G".

221. **Noord**, Gertjan van. The intersection of Finite State Automata and Definite Clause Grammars. In *33rd Annual Meeting of the Association for Computational Linguistics*, pages 159–165, June 1995. Mainly about DCGs, but contains a short but useful introduction to parsing as intersection.

222. **Albro**, Daniel M. Taking primitive optimality theory beyond the finite state. Technical report, Linguistics Department UCLA, 2000. Primitive optimality theory concerns the creation of a set of acceptable surface representations from an infinite source of underlying representations; acceptability is defined by a series of constraints. The set of representations is implemented as an FS machine, the constraints as weighted FS machines. The representation M generated by the infinite source is passed through each of the constraint machines and the weights are the penalties it incurs. After each constraint machine, all non-optimal paths are removed from M. All this can be done very efficiently, since FS machines can be intersected easily.
The paper proposes to increase the power of this system by allowing CF and multiple context-free grammars (Seki et al. [272]) as the representation; intersection with the constraining FS machines is still possible. It is trivial to extend an Earley parser to do this intersection job, but it just yields a set of sets of items, and a 50-lines algorithm to retrieve the new intersection grammar from these data is required and given in the paper. Further extensions of the intersecting Earley parser include the handling of the weights and the adaptation to MCF grammars. Fairly simple techniques suffice in all three cases.

18.2.5 Parallel Parsing Techniques

223. **Fischer**, Charles N. On parsing context-free languages in parallel environments. Technical Report 75-237, Cornell University, Ithaca, New York, April 1975. Introduces the parallel parsing technique discussed in Section 14.2. Similar techniques are applied for LR parsing and precedence parsing, with much theoretical detail.

224. **Brent**, R. P. and Goldschlager, L. M. A parallel algorithm for context-free parsing. *Australian Comput. Sci. Commun.*, 6(7):7.1–7.10, 1984. Is almost exactly the algorithm by Rytter [226], except that its recognize phase also does a plain CYK combine step, and their propose step is a bit more complex. Also proves $^2 \log n$ efficiency on $O(n^6)$ nodes. Suggests that it can be done on $O(n^4.9911)$ nodes, depending on Boolean matrix multiplication of matrices of size $O(n^2) \times O(n^2)$.

225. **Bar-On**, Ilan and Vishkin, Uzi. Optimal parallel generation of a computation tree form. *ACM Trans. Prog. Lang. Syst.*, 7(2):348–357, April 1985. Describes an optimal parallel algorithm to find a computation tree form from a general arithmetic expression. The heart of this algorithm consists of a parentheses-matching phase which solves this problem in time $O(\log\ n)$ using $n/\log\ n$ processors, where n is the number of symbols in the expression. First, the expression is split up into $n/\log\ n$ successive segments of length $\log\ n$, and each segment is assigned to a processor. Each processor then finds the pairs of matching parentheses in its own segment using a stack. This takes $O(\log\ n)$ time. Next, a binary tree is used to compute the nesting level of the left-over parentheses, and this tree is used to quickly find matching parentheses.

226. **Rytter**, Wojciech. On the recognition of context-free languages. In Andrzej Skowron, editor, *Computation Theory*, volume 208 of *Lecture Notes in Computer Science*, pages 318–325. Springer Verlag, Berlin, 1985. Describes the Rytter chevrons, which are represented as a pair of parse trees: the pair $((A,i,j),(B,k,l))$ is "realizable" if $A \stackrel{*}{\rightarrow} w[i\ldots k]Bw[l\ldots j]$, where $w[1\ldots n]$ is the input. The author then shows that using these chevrons, one can do CFL recognition in $O(\log^2 n)$ time on certain parallel machines using $O(n^6)$ processors; the dependence on the grammar size is not indicated. The paper also shows that the algorithm can be simulated on a multihead 2-way deterministic pushdown automaton in polynomial time.

227. **Rytter**, Wojciech. Parallel time $O(\log n)$ recognition of unambiguous CFLs. In *Fundamentals of Computation Theory*, volume 199 of *Lecture Notes in Computer Science*, pages 380–389. Springer Verlag, Berlin, 1985. Uses the Rytter chevrons as also described in [226] and shows that the resulting recognition algorithm can be executed in $O(\log n)$ time on a parallel W-RAM, which is a parallel random access machine which allows simultaneous reads and also simultaneous writes provided that the same value is written.

228. **Chang**, J. H., Ibarra, O. H., and Palis, M. A. Parallel parsing on a one-way array of finite-state machines. *IEEE Trans. Comput.*, 36:64–75, 1987. Presents a very detailed description of an implementation of the CYK algorithm on a one-way two-dimensional array of finite-state machines, or rather a 2-DSM, in linear time.

229. **Yonezawa**, Akinori and Ohsawa, Ichiro. Object-oriented parallel parsing for context-free grammars. In *COLING-88: 12th International Conference on Computational Linguistics*, pages 773–778, Aug. 1988. The algorithm is distributed bottom-up. For each rule $A \rightarrow BC$, there is an agent (object) which receives messages containing parse trees for Bs and Cs which have just been discovered, and, if the right end of B and the left end of C are adjacent, constructs a parse tree for A and sends it to every agent who manages a rule that has A as its RHS. ε-rules and circularities are forbidden.

230. **Srikant**, Y. N. Parallel parsing of arithmetic expressions. *IEEE Trans. Comput.*, 39(1):130–132, 1990. This short paper presents a parallel parsing algorithm for arithmetic expressions and analyzes its performance on different types of models of parallel computation. The parsing algorithm works in 4 steps:

1. Parenthesize the given expression fully.
2. Delete redundant parameters.
3. Separate the sub-expressions at each level of parenthesis nesting and determine the root of the tree form of each sub-expression in parallel.
4. Separate the sub-expressions at each level of parenthesis nesting and determine the children of each operator in the tree form of each sub-expression in parallel.

The algorithm takes $O(\sqrt{n})$ on a mesh-connected computer, and $O(\log^2 n)$ on other computation models.

231. **Alblas**, Henk, Nijholt, Anton, Akker, Rieks op den, Oude Luttighuis, Paul, and Sikkel, Klaas. An annotated bibliography on parallel parsing. Technical Report INF 92-84, University of Twente, Enschede, The Netherlands, Dec. 1992. Introduction to parallel parsing covering: lexical analysis, parsing, grammar decomposition, string decomposition, bracket matching, miscellaneous methods, natural languages, complexity, and parallel compilation; followed by an annotated bibliography of about 200 entries.

232. **Janssen**, W., Poel, M., Sikkel, K., and Zwiers, J. The primordial soup algorithm: A systematic approach to the specification of parallel parsers. In *Fifteenth International Conference on Computational Linguistics*, pages 373–379, Aug. 1992. Presents a general framework for specifying parallel parsers. The soup consists of partial parse trees that can be arbitrarily combined. Parsing algorithms can be described by specifying constraints in the way trees can be combined. The paper describes the mechanism for a.o. CYK and bottom-up Earley (BUE), which is Earley parsing without the top-down filter. Leaving out the top-down filter allows for parallel bottom-up, rather than left-to-right processing. The mechanism allows the specification of parsing algorithms without specifying flow control or data structures, which gives an abstract, compact, and elegant basis for the design of a parallel implementation.

233. **Sikkel**, Klaas and Lankhorst, Marc. A parallel bottom-up Tomita parser. In Günther Görz, editor, *1. Konferenz "Verarbeitung Natürlicher Sprache" - KONVENS'92*, Informatik Aktuell, pages 238–247. Springer-Verlag, Oct. 1992. Presents a parallel bottom-up GLR parser that can handle any CF grammar. Removes the left-to-right restriction and introduces processes that parse the sentence, starting at every position in the input, in parallel. Each process yields the parts that start with its own word. The processes are organized in a pipeline. Each process sends the completed parts that it finds and the parts that it receives from his right neighbor to his left neighbor, who combines the parts that it receives with the parts that it already found to create new parts. It uses a simple pre-computed parsing table and a graph-structured stack (actually tree-structured) in which (partially) recognized parts are stored. Empirical results indicate that parallelization pays off for sufficiently long sentences, where "sufficiently long" depends on the grammar. A sequential Tomita parser is faster for short sentences. The algorithm is discussed in Section 14.3.1.

234. **Alblas**, Henk, Akker, Rieks op den, Oude Luttighuis, Paul, and Sikkel, Klaas. A bibliography on parallel parsing. *ACM SIGPLAN Notices*, 29(1):54–65, 1994. A modified and compacted version of the bibliography by Alblas et al. [231].

235. **Hendrickson**, Kenneth J. A new parallel LR parsing algorithm. In *ACM Symposium on Applied Computing*, pages 277–281. ACM, 1995. Discusses the use of a *marker-passing* computational paradigm for LR parsing. Each state in the LR parsing table is modeled as a node with links to other nodes, where the links represent state transitions. All words in the input sentences are broadcast to all nodes in the graph, acting as *activation* markers. In addition, each node has a *data* marker specifying which inputs are legal for shifting a token and/or which reduction to use. The parsing process is then started by placing an initial *prediction* marker for each sentence on the start node. When a prediction marker arrives at a node, it will collide with the activation markers at that node, provided they are at the same position in the same sentence. The result of such a collision is determined by the data marker at that node which may specify reductions and/or shifts, which are handled sequentially, resulting in new prediction markers which are sent to their destination node.

236. **Ra**, Dong-Yul and Kim, Jong-Hyun. A parallel parsing algorithm for arbitrary context-free grammars. *Inform. Process. Lett.*, 58(2):87–96, 1996. A parallel parsing algorithm based on Earley's algorithm is proposed. The Earley items construction phase is parallelized by assigning a processor to each position in the input string. Each processor i then performs n stages: stage k consists of the computation of all Earley items of "length" k which start at position i. After each stage, the processors are synchronized, and items are transferred. It turns out that this only requires data transfer from processor $i+1$ to processor i. Any items that processor i needs from processor $i+m$ are obtained by processor $i+1$ at stage $m-1$. When not enough processors are available ($p < n$), a stage is divided into $\lceil n/p \rceil$ phases, such that processor i computes all items starting at positions i, $i+p$, $i+2p$, et cetera. If in the end processor 0 found an item $\texttt{S} \rightarrow \alpha \bullet \texttt{,0,}n$, the input string is recognized.
To find a parse, each processor processes requests to find a parse for completed items (i.e. the dot is at the end of the right hand side) that it created. In processing such a request, the processor generates requests to other processors. Now processor 0 is asked $\texttt{S} \rightarrow \alpha \bullet \texttt{,0,}n$.
A very detailed performance analysis is given, which shows that the worst case performance of the algorithm is $O(n^3/p)$ on p processors.

18.2.6 Non-Chomsky Systems

237. **Koster**, Cornelis H. A. and Meertens, Lambert G. L. T. Basic English, a generative grammar for a part of English. Technical report, Euratom Seminar "Machine en Talen" of E.W. Beth, University of Amsterdam, 1962. [2]

238. **McClure**, R. M. TMG: A syntax-directed compiler. In *20th National Conference*, pages 262–274. ACM, 1965. A transformational grammar system in which the syntax is described by a sequence of parsing routines, which can succeed, and then may absorb input and produce output, or fail, and then nothing has happened; this requires backtracking. Each routine consists of a list of possibly labeled calls to other parsing routines of the form <*routine_name/failure_label*>. If the called routine succeeds, the next call in the list is performed; if it fails, control continues at the *failure_label*. An idiom for handling left recursion is given. This allows concise formulation of many types of input. Rumor has it that TMG stands for "transmogrify", but "transformational grammar" is equally probable.

239. **Gilbert**, Philip. On the syntax of algorithmic languages. *J. ACM*, 13(1):90–107, Jan. 1966. Unlike Chomsky grammars, which are production devices, an "analytic grammar" is a recognition device: membership of the language is decided by an algorithm based on the analytic grammar. An analytic grammar is a set of reduction rules, which are Chomsky Type 1 production rules in reverse, plus a scan function S. An example of a reduction rule is $\texttt{abcde} \rightarrow \texttt{aGe}$, which reduces $\texttt{bcd}$ to $\texttt{G}$ in the context $\texttt{a} \cdots \texttt{e}$.
A string belongs to the language if it can be reduced to the start symbol by applying reduction rules, such that the position of each reduction in the sentential form is allowed by the scan function. Reduction can never increase the length of the sentential form, so if we avoid duplicate sentential forms, this process always terminates. So an analytic grammar recognizes a recursive set. The author also proves that for every recursive set there is analytic grammar which recognizes it; this may require complicated scan functions.
Two examples are given: Hollerith constants, and declaration and use of identifiers. There seem to be no further publications on analytic grammars.

240. **Hotz**, G. Erzeugung formaler Sprachen durch gekoppelte Ersetzungen. In F.L. Bauer and K. Samelson, editors, *Kolloquium über Automatentheorie und formale Sprachen*, pages 62–73. TU Munich, 1967, (in German). Terse and cryptic paper in which the components of Chomsky's grammars and its mechanism are generalized into an X-category, consisting of

[2] It is to be feared that this paper is lost. Any information to the contrary would be most welcome.

an infinite set of sentential forms, a set of functions that perform substitutions, a set of sources (left-hand sides of any non-zero length), a set of targets (right-hand sides of any length), an inference operator (for linking two substitutions), and a concatenation operator. By choosing special forms for the functions and the operators and introducing a number of homomorphisms this mechanism is used to define coupled production. Using theorems about X-categories it is easy to prove that the resulting languages are closed under union, intersection and negation. Many terms not explained; no examples given.

241. **Sintzoff**, M. Existence of a van Wijngaarden syntax for every recursively enumerable set. *Annales de la Société Scientifique de Bruxelles*, 81(II):115–118, 1967. A relatively simple proof of the theorem that for every semi-Thue system we can construct a VW grammar that produces the same set.

242. **Friš**, Ivan. Grammars with partial ordering of the rules. *Inform. Control*, 12:415–425, 1968. The CF grammars are extended with restrictions on the production rules that may be applied in a given production step.
One restriction is to have a partial order on the production rules and disallow the application of a production rule if a smaller (under the partial ordering) rule could also be applied. This yields a language class in between CF and CS which includes $a^n b^n c^n$, but the author uses 26(!) rules and 15 orderings to pull this off.
Another restriction is to require the control word of the derivation to belong to a given regular language. This yields exactly the CS languages. A formal proof is given, but no example.
For errata see "Inform. Control", 15(5):452-453, Nov. 1969.
(The *control word* of a derivation *D* is the sequence of the numbers of the production rules used in *D*, in the order of their application. The term is not used in this paper and is by Salomaa.)

243. **Knuth**, Donald E. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968. Introduces inherited attributes after acknowledging that synthesized attributes were already used by Irons in 1961. Shows how inherited attributes may simplify language description, mainly by localizing global effects. Gives a formal definition of attribute grammars and shows that they can express any expressible computation on the parse tree, by carrying around an attribute that represents the entire tree.
With having both synthesized and inherited attributes comes the danger of circularity of the attribute rules. An algorithm is given to determine that situation statically (corrected by the author in *Math. Syst. Theory*, **5**, 1, 1971, pp. 95-96.)
Next a simple but non-trivial language for programming a Turing machine called Turingol is defined using an attribute grammar. The full definition fits on one printed page. A comparison with other systems (Vienna Definition Language, etc.) concludes the paper.

244. **Wijngaarden**, A. van et al. Report on the algorithmic language ALGOL 68. *Numer. Math.*, 14:79–218, 1969. VW grammars found their widest application to date in the definition of ALGOL 68. Section 1.1.3 of the ALGOL 68 Revised Report contains a very carefully worded description of the two-level mechanism. The report contains many interesting applications.
See also [251].

245. **Koster**, C. H. A. Affix grammars. In J.E.L. Peck, editor, *ALGOL 68 Implementation*, pages 95–109. North-Holland Publ. Co., Amsterdam, 1971. Where attribute grammars have attributes, affix grammars have affixes, and where attribute grammars have evaluation functions affix grammars have them too, but the checks in an affix grammar are part of the grammar rather than of the evaluation rules. They take the form of primitive predicates, pseudo-non-terminals with affixes similar to the `where...` predicates in a VW grammar, which produce ε when they succeed, but block the production process when they fail. Unlike attribute grammars, affix grammars are production systems. If the affix grammar is "well-formed", a parser for it can be constructed.

246. **Birman**, Alexander and Ullman, Jeffrey D. Parsing algorithms with backtrack. *Inform. Control*, 23(1):1–34, 1973. Whereas a Chomsky grammar is a mechanisms for generating languages, which can, with considerable difficulty, be transformed into a parsing mechanism, a *TS* (*TMG recognition Scheme*), (McClure [238]) is a top-down parsing technique, which can, with far

less difficulty, be transformed into a language generation mechanism. Strings that are accepted by a given TS belong to the language of that TS.
A TS is a set of recursive routines, each of which has the same structure: *A* **`= if`** *recognize B* **`andif`** *recognize C* **`then`** *succeed* **`else`** *recognize D* **`fi`**, where each routine does backtracking when it returns failure; this models backtracking top-down parsing. This routine corresponds to the TS rule $A \rightarrow BC/D$.
The paper also introduces *generalized TS* (*gTS*), which has rules of the form $A \rightarrow B(C,D)$, meaning *A* **`= if`** *recognize B* **`then`** *recognize C* **`else`** *recognize D* **`fi`**. This formalism allows negation: **`return if`** *recognize* A_i **`then`** *fail* **`else`** *succeed* **`fi`**.
TS and gTS input strings can be recognized in one way only, since the parsing algorithm is just a deterministic program. TS and gTS languages can be recognized in linear time, as follows. There are $|V|$ routines, and each can be called in $n+1$ positions, where V is the set of non-terminals and n is the length of the input string. Since the results of the recognition routines depend only on the position at which they are started, their results can be precomputed and stored in a $|V| \times n$ matrix. A technique is shown by which this matrix can be computed from the last column to the first.
Since CF languages probably cannot be parsed in linear time, there are probably CF languages which are not TS or gTS, but none are known. [g]TS languages are closed under intersection (recognize by one TS, fail, and then recognize by the other TS), so there are non-CF languages which are [g]TS; $a^n b^n c^n$ is an example. Many more such properties are derived and proved in a heavy formalism.

247. **Lepistö**, Timo. On ordered context-free grammars. *Inform. Control*, 22(1):56–68, Feb. 1973. More properties of ordered context-free grammars (see Friš [242]) are given.

248. **Schuler**, P. F. Weakly context-sensitive languages as model for programming languages. *Acta Inform.*, 3(2):155–170, 1974. *Weakly context-sensitive languages* are defined in two steps. First some CF languages are defined traditionally. Second a formula is given involving the CF sets, Boolean operators, quantifiers, and substitutions; this formula defines the words in the WCS language. An example is the language $\mathcal{L}_0 = \mathbf{a}^n\mathbf{b}^n\mathbf{a}^n$. We define the CF languages $S_1 = \mathbf{a}^n\mathbf{b}^n$ and $S_2 = \mathbf{a}^k$. Then $\mathcal{L}_0 = \{w | \exists x \in S_1 \exists y \in S_2 | xy = w \wedge \exists z | y\mathbf{b}z = x\}$. It is shown that this is stronger than CF but weaker than CS. WCS languages are closed under union, intersection, complementation and concatenation, but not under unbounded concatenation (Kleene star). A Turing machine parser is sketched, which recognizes strings in $O(n^k)$ where k depends on the complexity of the formula. A WCS grammar is given, which checks definition and application of variables and labels in ALGOL 60. The unusual formalism and obscure text make the paper a difficult read.

249. **Wijngaarden**, A. van. The generative power of two-level grammars. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 9–16. Springer-Verlag, Berlin, 1974. The generative power of VW grammars is illustrated by creating a VW grammar that simulates a Turing machine; the VW grammar uses only one metanotion, thus proving that one metanotion suffices.

250. **Joshi**, Aravind K., Levy, Leon S., and Takahashi, Masako. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, 1975. See Section 15.4.
The authors start by giving a very unintuitive and difficult definition of trees and tree grammars, which fortunately is not used in the rest of the paper. A hierarchy of tree adjunct grammars is constructed, initially based on the maximum depth of the adjunct trees. This hierarchy does not coincide with Chomsky's:
$$L(TA(1)) \subset L(CF) \subset L(TA(2)) \subset L(TA(3)) \subset \cdots \subset L(CS)$$
A "simultaneous tree adjunct grammar" (STA grammar) also consists of a set of elementary trees and a set of adjunct trees, but the adjunct trees are divided into a number of groups. In each adjunction step one group is selected, and all adjunct trees in a group must be applied simultaneously. It is shown that:
$$L(CF) \subset L(TA(n)) \subset L(STA) \subset L(CS)$$

251. **Wijngaarden**, A. van et al. Revised report on the algorithmic language ALGOL 68. *Acta Inform.*, 5:1–236, 1975. See van Wijngaarden et al. [244].

252. **Cleaveland**, J. Craig and Uzgalis, Robert C. *Grammars for Programming Languages*. Elsevier, New York, 1977. In spite of its title, the book is a highly readable explanation of two-level grammars, also known as *van Wijngaarden grammars* or VW grammars. After an introductory treatment of formal languages, the Chomsky hierarchy and parse trees, it is shown to what extent CF languages can be used to define a programming language. These are shown to fail to define a language completely and the inadequacy of CS grammars is demonstrated. VW grammars are then explained and the remainder of the book consists of increasingly complex and impressive examples of what a VW grammar can do. These examples include keeping a name list, doing type checking and handling block structure in the definition of a programming language. Recommended reading.

253. **Meersman**, R. and Rozenberg, G. Two-level meta-controlled substitution grammars. *Acta Inform.*, 10:323–339, 1978. The authors prove that the uniform substitution rule is essential for two-level grammars; without it, they would just generate the CF languages. This highly technical paper examines a number of variants of the mechanisms involved.

254. **Dembiński**, Piotr and Małuszyński, Jan. Two-level grammars: CF grammars with equation schemes. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, volume 71 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, Berlin, 1979. The authors address a restricted form of VW grammars in which each metanotion produces a regular language and no metanotion occurs more than once in a hypernotion; such grammars still have full Type 0 power. A context-free skeleton grammar is derived from such a grammar by brute force: each hypernotion in the grammar is supposed to produce each other hypernotion, through added renaming hyperrules. Now the context-free structure of the input is handled by the skeleton grammar whereas the context conditions show up as equations derived trivially from the renaming rules.
The equations are string equations with variables with regular domains. To solve these equations, first all variables are expressed in a number of new variables, each with the domain Σ^*. Then each original variable is restricted to its domain. Algorithms in broad terms are given for both phases.
Any general context-free parser is used to produce all parse trees and for each parse tree we try to solve the set of equations corresponding to it. If the attempt succeeds, we have a parsing. This process will terminate if the skeleton grammar identifies a finite number of parse trees, but in the general case the skeleton grammar is infinitely ambiguous and we have no algorithm.

255. **Kastens**, Uwe. Ordered attribute grammars. *Acta Inform.*, 13(3):229–256, 1980. A *visit* to a node is a sequence of instructions of two forms: evaluate attribute m of child n or of the parent, and perform visit k of child n. A node may require more than one visit, hence the "visit k". If a sequence of visits exists for all nodes so that all attributes are evaluated properly, which is almost always the case, the attribute grammar is *ordered*.

256. **Wegner**, Lutz Michael. On parsing two-level grammars. *Acta Inform.*, 14:175–193, 1980. The article starts by defining a number of properties a VW grammar may exhibit; among these are "left-bound", "right-bound", "free of hidden empty notions", "uniquely assignable" and "locally unambiguous". Most of these properties are undecidable, but sub-optimal tests can be devised. For each VW grammar G_{VW}, a CF *skeleton grammar* G_{SK} is defined by considering all hypernotions in the VW grammar as non-terminals of G_{SK} and adding the cross-references of the VW grammar as production rules to G_{SK}. G_{SK} generates a superset of G_{VW}. The cross-reference problem for VW grammars is unsolvable but again any sub-optimal algorithm (or manual intervention) will do. Parsing is now done by parsing with G_{SK} and then reconstructing and testing the metanotions. A long list of conditions necessary for the above to work are given; these conditions are in terms of the properties defined at the beginning.

257. **Wijngaarden**, A. van. Languageless programming. In *IFIP/TC2/WG2.1 Working Conference on the Relations Between Numerical Computation and Programming Languages*, pages 361–371. North-Holland Publ. Comp., 1981. Forbidding-looking paper which presents an interpreter for a stack machine expressed in a VW grammar. The paper is more accessible than it would seem: the interpreter "reads" — if the term applies — as a cross between Forth and assembler language. A simple but non-trivial program, actually one hyperrule, is given,

which computes the n-th prime, subtracts 25 and "outputs" the answer in decimal notation. The interpreter and the program run correctly on Grune's interpreter [260].

258. **Gerevich**, László. A parsing method based on van Wijngaarden grammars. *Computational Linguistics and Computer Languages*, 15:133–156, 1982. In consistent substitution, a metanotion is replaced consistently by one of its terminal productions; in *extended consistent substitution*, a metanotion is replaced consistently by one of the sentential forms it can produce. The author proves that VW grammars with extended consistent substitution are equivalent to those with just consistent substitution; this allows "lazy" evaluation of the metanotions during parsing. Next an example of a top-down parser using the lazy metanotions as logic variables (called here "grammar-type variables") is shown and demonstrated extensively. The third part is a reasonably intuitive list of conditions under which this parser type works, presented without proof. The fourth part shows how little the VW grammar for a small ALGOL 68-like language needs to be changed to obey these conditions.

259. **Watt**, D. A. and Madsen, O. L. Extended attribute grammars. *Computer J.*, 26(2):142–149, 1983. The assignment rules $A_i := f_i(A_j,\ldots,A_k)$ of Knuth's [243] are incorporated into the grammar by substituting $f_i(A_j,\ldots,A_k)$ for A_i. This allows the grammar to be used as a production device: production fails if any call is undefined. The grammar is then extended with a transduction component; this restores the semantics expressing capability of attribute grammars. Several examples from compiler construction given.

260. **Grune**, Dick. How to produce all sentences from a two-level grammar. *Inform. Process. Lett.*, 19:181–185, Nov. 1984. All terminal productions are derived systematically in breadth-first order. The author identifies pitfalls in this process and describes remedies. A parser is used to identify the hyperrules involved in a given sentential form. This parser is a general CF recursive descent parser to which a consistency check for the metanotions has been added; it is not described in detail.

261. **Małuszyński**, J. Towards a programming language based on the notion of two-level grammar. *Theoret. Comput. Sci.*, 28:13–43, 1984. In order to use VW grammars as a programming language, the cross-reference problem is made solvable by requiring the hypernotions to have a tree structure rather than be a linear sequence of elements. It turns out that the hyperrules are then a generalization of the Horn clauses, thus providing a link with DCGs.

262. **Edupuganty**, Balanjaninath and Bryant, Barrett R. Two-level grammars for automatic interpretation. In *1985 ACM Annual Conference*, pages 417–423. ACM, 1985. First the program is parsed without regard to the predicate hyperrules; this yields both instantiated and uninstantiated metanotions. Using unification-like techniques, these metanotions are then checked in the predicates and a set of interpreting hyperrules is used to construct the output metanotion. All this is similar to attribute evaluation. No exact criteria are given for the validity of this procedure, but a substantial example is given.
The terminal symbols are not identified separately but figure in the hypernotions as protonotions; this is not fundamental but does make the two-level grammar more readable.

263. **Fisher**, A. J. Practical LL(1)-based parsing of van Wijngaarden grammars. *Acta Inform.*, 21:559–584, 1985. Fisher's parser is based on the idea that the input string was generated using only a small, finite, part of the infinite *strict grammar* that can be generated from the VW grammar. The parser tries to reconstruct this part of the strict grammar on the fly while parsing the input. The actual parsing is done by a top-down interpretative LL(1) parser, called the *terminal parser*. It is driven by a fragment of the strict grammar and any time the definition of a non-terminal is found missing by the terminal parser, it asks another module, the *strict syntax generator*, to try to construct it from the VW grammar. For this technique to work, the VW grammar has to satisfy three conditions: the defining CF grammar of each hyperrule is unambiguous, there are no free metanotions, and the skeleton grammar (as defined by Wegner [256]) is LL(1). The parser system is organized as a set of concurrent processes (written in occam), with both parsers, all hyperrule matchers and several other modules as separate processes. The author claims that "this concurrent organization ... is strictly a property of the algorithm, not of the implementation", but

a sequential, albeit slower, implementation seems quite possible. The paper gives heuristics for the automatic generation of the cross-reference needed for the skeleton grammar; gives a method to handle *general hyperrules*, hyperrules that fit all hypernotions, efficiently; and pays much attention to the use of angle brackets in VW grammars.

264. **Vijay-Shankar**, K. and Joshi, Aravind K. Some computational properties of tree adjoining grammars. In *23rd Annual Meeting of the ACL*, pages 82–93, University of Chicago, Chicago, IL, July 1985. Parsing: the CYK algorithm is extended to TAGs as follows. Rather than having a two-dimensional array $A_{i,j}$ the elements of which contain non-terminals that span $t_{i..j}$ where t is the input string, we have a four-dimensional array $A_{i,j,k,l}$ the elements of which contain tree nodes X that span $t_{i..j}..t_{k..l}$, where the gap $t_{j+1,k-1}$ is spanned by the tree hanging from the foot node of X. The time complexity is $O(n^6)$ for TAGs that are in "two form" .
Properties: informal proofs are given that TAGs are closed under union, concatenation, Kleene star, and intersection with regular languages.

265. **Barnard**, D. T. and Cordy, J. R. SL parses the LR languages. *Comput. Lang.*, 13(2):65–74, July 1988. *SL* (*Syntax Language*) is a special-purpose language for specifying recursive input-output transducers. An SL program consists of a set of recursive parameterless routines. The code of a routine can call other routines, check the presence of an input token, produce an output token, and perform an n-way switch on the next input token, which gets absorbed in the process. Blocks can be repeated until an exit statement is switched to. The input and output streams are implicit and are the only variables.

266. **Schabes**, Yves and Vijay-Shankar, K. Deterministic left to right parsing of tree adjoining languages. In *28th Meeting of the Association for Computational Linguistics*, pages 276–283. Association for Computational Linguistics, 1990. Since production using a TAG can be based on a stack of stacks (see Vijay-Shankar and Joshi [264]), the same model is used to graft LR parsing on. Basically, the stacks on the stack represent the reductions of the portion left of the foot in each adjoined tree; the stack itself represents the spine of the entire tree recognized so far. Dotted trees replace the usual dotted items; stack manipulation during the "Resume Right" operation, basically a shift over a reduced tree root, is very complicated. See Nederhof [281].

267. **Heilbrunner**, S. and Schmitz, L. An efficient recognizer for the Boolean closure of context-free languages. *Theoret. Comput. Sci.*, 80:53–75, 1991. The CF grammars are extended with two operators: negation (anything not produced by A) and intersection (anything produced by both A and B). The non-terminals in the grammar have to obey a hierarchical order, to prevent paradoxes: $A \rightarrow A$ would define an A which produces anything not produced by A. An Earley parser in CYK formulation (Graham et al. [23]) is extended with inference (dot-movement) rules for these operators, and a special computation order for the sets is introduced. This leads to a "naive" (well...) algorithm, to which various optimizations are applied, resulting in an efficient $O(n^3)$ algorithm. A 10-page formal proof concludes the paper.

268. **Koster**, C. H. A. Affix grammars for natural languages. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 469–484, New York, 1991. Springer Verlag. The domains of the affixes are restricted to finite lattices, on the grounds that this is convenient in linguistics; lattices are explained in the text. This formally reduces the grammar to a CF one, but the size can be spectacularly smaller. Inheritance and subsetting of the affixes is discussed, as are parsing and left recursion. An example for English is given. Highly amusing account of the interaction between linguist and computer scientist.

269. **Koster**, C. H. A. Affix grammars for programming languages. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 358–373. Springer-Verlag, New York, 1991. After a historical introduction, the three formalisms VW Grammar, Extended Attribute/Affix Grammar, and Attribute Grammar are compared by implementing a very simple language consisting of declarations and assignments in them. The comparison includes Prolog. The conclusion finds

far more similarities than differences; VW Grammars are the most descriptional, Attribute Grammars the most operational, with EAGs in between.

270. **Krulee**, Gilbert K. *Computer Processing of Natural Language*. Prentice-Hall, 1991. Concentrates on those classes of grammars and features that are as applicable to English as to Pascal (paraphrase of page 1). No overly soft linguistic talk, no overly harsh formalisms.
The book is based strongly on two-level grammars, but these are not the Van Wijngaarden type in that no new non-terminals are produced. The metanotions produce strings of terminals and non-terminals of the CF grammar rather than segments of names of non-terminals. When this is done, the applied occurrences of metanotions in the CF grammar must be substituted using some uniform substitution rule. An Earley parser for this kind of two-level grammars is sketched. Parsers for ATN systems are also covered.

271. **Schabes**, Yves and Joshi, Aravind K. Parsing with lexicalized tree adjoining grammars. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, pages 25–47. Kluwer Academic Publ., Boston, 1991. A grammar is "lexicalized" if each right-hand side in it contains at least one terminal, called its "anchor". Such grammars cannot be infinitely ambiguous. In parsing a sentence from a lexicalized grammar, one can first select the rules that can play a role in parsing, based on the terminals they contain, and restrict the parser to these. In very large grammars this helps.
Various parser variants for this structure are described: CYK, top-down, Earley and even LR.
Feature-based tree adjoining grammars are tree adjoining grammars with attributes and unification rules attached to each node. Although recognition for feature-based tree adjoining grammars is undecidable, an adapted Earley algorithm is given that will parse a restricted set of feature-based lexicalized tree adjoining grammars.

272. **Seki**, Hiroyuki, Matsumura, Takashi, Fuji, Mamoru, and Kasami, Tadao. On multiple context-free grammars. *Theoret. Comput. Sci.*, 88:191–229, 1991. Each non-terminal in a *multiple context-free grammar* (MCFG) produces a fixed number of strings rather than just one string; so it has a fixed number of right-hand sides. Each right-hand side is composed of terminals and components of other non-terminals, under the condition that if a component of a non-terminal A occurs in the right-hand side of a non-terminal B all components of A must be used. Several varieties are covered in the paper, each with slightly different restrictions.
MCFGs are stronger than CFGs: for example, $S \rightarrow (aS_1, bS_2, cS_3) | (\varepsilon, \varepsilon, \varepsilon)$, where S_1, S_2, and S_3 are the components of S, produces the language $a^n b^n c^n$. But even the strongest variety is weaker than CS.
Properties of this type of grammars are derived and proved; the grammars themselves are written in a mathematical notation. An $O(n^e)$ recognition algorithm is given, where e is a grammar-dependent constant. The algorithm is a variant of CYK, in that it constructs bottom-up sets of components of increasing length, until that length is equal to the length of the input. Parsing (the recovery of the derivation tree) is not discussed.

273. **Fisher**, Anthony J. A "yo-yo" parsing algorithm for a large class of van Wijngaarden grammars. *Acta Inform.*, 29(5):461–481, 1992. High-content paper describing a top-down parser which tries to reconstruct the production process that led to the input string, using an Earley-style parser to construct metanotions bottom-up where needed; it does not involve a skeleton grammar. It can handle a class of VW grammars characterized roughly by the following conditions: the cross-reference problem must be solvable by LL(1) parsing of the hypernotions; certain mixes of "left-bound" and "right-bound" (see Wegner [256]) do not occur; and the VW grammar is not left-recursive. The time requirement is $O(n^3 f^3(n))$, where $f(n)$ depends on the growth rate of the fully developed hypernotions ("protonotions") as a function of the length of the input. For "decent" grammars, $f(n) = n$, and the time complexity is $O(n^6)$.

274. **Grune**, Dick. Two-level grammars are more expressive than Type 0 grammars — or are they?. *ACM SIGPLAN Notices*, 28(8):43–45, Aug. 1993. VW grammars can construct names of non-terminals, but they can equally easily construct names of terminals, thus allowing the grammar to create new terminal symbols. This feat cannot be imitated by Type 0 grammars, so in a

sense VW grammars are more powerful. The paper gives two views of this situation, one in which the statement in the title is true, and one in which it is undefined.

275. **Pitsch**, Gisela. *LL(k)* parsing of coupled context-free grammars. *Computational Intelligence*, 10(4):563–578, 1994. LL parsing requires the prediction of a production $A \rightarrow A_1, A_2, \ldots, A_n$, based on look-ahead, and in CCFG we need the look-ahead at n positions in the input. Although we know which position in the input corresponds to A_1, we do not know which positions match $A_2, \ldots, A_n$, and we cannot obtain the required look-aheads. We therefore restrict ourselves to strong LL, based on the FIRST set of A_1 and the FOLLOW sets of $A_1, \ldots, A_n$. Producing the parse tables is complex, but parsing itself is simple, and linear-time.

276. **Satta**, Giorgio. Tree adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics*, 20(2):173–192, 1994. Proves that if we can do tree parsing in $O(n^p)$, we can do Boolean matrix multiplication in $O(n^{2+p/6})$, which for $p = 6$ amounts to the standard complexities for both processes. Since Boolean matrix multiplication under $O(n^3)$ is very difficult, it is probable that tree parsing under $O(n^6)$ is also very difficult.

277. **Pitsch**, Gisela. LR(*k*)-coupled-context-free grammars. *Inform. Process. Lett.*, 55(6):349–358, Sept. 1995. The coupling between the components of the coupled non-terminals is implemented by adding information about the reduction of a component X_1 to a list called "future", which runs parallel to the reduction stack. This list is used to control the LR automaton so that only proper reduces of the further components X_n of X will occur.

278. **Hotz**, Günter and Pitsch, Gisela. On parsing coupled-context-free languages. *Theoret. Comput. Sci.*, 161(1-2):205–233, 1996. General parsing with CCF grammars, mainly based on the CYK algorithm. Full algorithms, extensive examples.

279. **Kulkarni**, Sulekha R. and Shankar, Priti. Linear time parsers for classes of non context free languages. *Theoret. Comput. Sci.*, 165(2):355–390, 1996. The non-context-free languages are generated by two-level grammars as follows. The rules of the base grammar are numbered and one member of each RHS is marked as distinguished; the start symbol is unmarked. So from each unmarked non-terminal in the parse tree one can follow a path downward by following marked non-terminals, until one reaches a terminal symbol. A parse tree is acceptable only if the sequence of numbers of the rules on each such path is generated by the control grammar.
LL(1) and LR(1) parsers for such grammars, using stacks of stacks, are described extensively.

280. **Rußmann**, A. Dynamic LL(*k*) parsing. *Acta Inform.*, 34(4):267–290, 1997. Theory of LL(1) parsing of dynamic grammars.

281. **Nederhof**, Mark-Jan. An alternative LR algorithm for TAGs. In *36th Annual Meeting of the Association for Computational Linguistics*, pages 946–952. ACL, 1998. The traditional LR parsing algorithm is extended in a fairly straightforward way to parsing TAGs. It uses the traditional LR stack containing states and symbols alternately, although the symbols are sometimes more complicated. The author shows that Schabes and Vijay-Shankar's algorithm [266] is incorrect, and recognizes incorrect strings.
Upon implementation, it turned out that the LR transition tables were "prohibitively large" (46MB) for a reasonable TAG for English. But the author represents the table as a set of Prolog clauses (!) and does not consider table compression.

282. **Prolo**, Carlos A. An efficient LR parser generator for tree adjoining grammars. In *6th Int. Workshop on Parsing Technologies (IWPT 2000)*, pages 207–218, 2000. Well-argued exposition of the problems inherent in LR parsing of TAGs. Presents an LR parser generator which produces tables that are one or two orders of magnitude smaller than Nederhof's [281], making LR parsing of tree adjoining grammars more feasible.

283. **Okhotin**, Alexander. Conjunctive grammars. *J. Automata, Languages and Combinatorics*, 6(4):519–535, 2001. A conjunctive grammar is a CF grammar with an additional intersection operation. Many properties of conjunctive grammars are shown and proven,

and many examples are provided. For example, the conjunctive grammars are stronger than the intersection of a finite number of CF languages. They lead to parse dags. Tabular parsing is possible in time $O(n^3)$.

284. **Ford**, Bryan. Packrat parsing: Simple, powerful, lazy, linear time. *ACM SIGPLAN Notices*, 37(9):36–47, Sept. 2002. A straightforward backtracking top-down parser in Haskell is supplied with memoization (see Section 17.3.4), which removes the need for repeated backtracking and achieves unbounded look-ahead. Linear-time parsing is achieved by always matching the largest possible segment; this makes the result of a recognition unique, and the parsing unambiguous. Left recursion has to be removed by the user, but code is supplied to produce the correct parse tree nevertheless. Since the memoized functions remember only one result and then stick to that, Packrat parsing cannot handle all CF languages; a delineation of the set of suitable languages is not given. See, however, Ford [286]. Implementation of the parser using monads is discussed.

285. **Jackson**, Quinn Tyler. Efficient formalism-only parsing of XML/HTML using the §-calculus. *ACM SIGPLAN Notices*, 38(2):29–35, Feb. 2003. The §-calculus is a CF grammar in which new values can be dynamically assigned to non-terminals in the grammar during parsing. Such values can be the value of a generic terminal (identifiers, etc.) found in the input or a new CF production rule, somewhat similar to the Prolog assert feature. This allows context-sensitive restrictions to be incorporated in the grammar. This system is used to write a concise grammar capable of handling both XML and HTML documents. It is then run on Meta-S, a backtracking LL(k) recursive descent parser for the §-calculus.

286. **Ford**, Bryan. Parsing expression grammars: A recognition-based syntactic foundation. In *31st ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 111–122. ACM, Jan. 2004. A *PEG* (*Parsing Expression Grammar*) describes a language by being a recognition algorithm. It is basically an EBNF grammar whose meaning is determined by a top-down interpreter, similar to those described by Birman and Ullman [246]. The interpreter works left-to-right top-to-bottom and always consumes the longest possible input: an expression $e_1 e_2 \cdots / e_3 \cdots$ means **if** e_1 **andif** e_2 **andif** $\cdots$ **then** *succeed* **else** e_3 **andif** $\cdots$ **fi**. If an expression succeeds it consumes what it has recognized; if an expression fails, it consumes nothing, even if subsections of it have recognized some input. This requires backtracking. PEGs have two additional operators, &A, which tests for the presence of an A but consumes nothing, and !A, which tests for the absence of an A and consumes nothing. PEGs have to be "well-formed", which basically means "not left-recursive".
PEGs have several advantages over CF grammars: PEGs are unambiguous; PEG languages are closed under intersection and negation; PEGs can recognize some non-CF languages; and parsing with PEGs can be done in linear time.
These and several other properties — static analysis, well-formedness, algebraic equalities, relation to Birman and Ullman's TS and gTS — are proved in the paper, with short common-sense proofs.

287. **Grimm**, Robert. Practical packrat parsing. Technical Report TR2004-854, Dept. of Computer Science, New York University, New York, March 2004. Describes an object-oriented implementation of Packrat parsing in Java, called Rats"!. It allows the attachment of semantics to rules.

288. **Okhotin**, Alexander. Boolean grammars. *Information and Computation*, 194(1):19–48, 2004. Boolean grammars are CF grammars extended with intersection and negation. The languages they define are not described by a substitution mechanism, but in one of two ways: as the solution of a set of equations, and as the partial fixed point of a function. It is not necessary for both of them to exist, but it is shown that *if* both exist, they define the same language. If neither solution exists, the grammar is not well-formed. Many properties of Boolean grammars are shown and proven; a binary form is defined; and the corresponding CYK algorithm is presented, yielding a parse dag. This allows parsing in $O(n^3)$. Remarkably they can be recognized in $O(n)$ space, but that takes some doing.
Linear Boolean grammars are Boolean grammars in which each conjunct contains at most one non-terminal. They are proven to be equivalent to trellis automata. Useful tables of comparisons of grammars and languages complete the paper.

289. **Okhotin**, Alexander. LR parsing for boolean grammars. In *International Conference on Developments in Language Theory (DLT)*, volume 9, pages 362–373, 2005. GLR parsing is extended with two operations, a "conjunctive reduce", which is almost the same as the traditional reduce, except that for $X \to ABC\&DEF$ it reduces only if both ABC and DEF are present, and an *invalidate*, which removes clusters of branches from the GSS in response to finding an X where the grammar calls for $\neg X$. Complete algorithms are given. The time complexity is $O(n^4)$, which can be reduced to $O(n^3)$ by further memoization. A short sketch of an LL(1) parser for Boolean grammars is also given.

290. **Okhotin**, Alexander. On the existence of a Boolean grammar for a simple procedural language. In *11th International Conference on Automata and Formal Languages: AFL'05*, 2005. A paradigm for using Boolean grammars for the formal specification of programming languages is being developed. The method involves a sublanguage $C = I\Sigma^*I$, where both occurrences of I represent the same identifier and Σ^* can be anything as long as it sets itself off against the two identifiers. The CF part of the Boolean grammar is then used to assure CF compliance of the program text, and repeated intersection with C is used to insure that all identifiers are declared and intersection with $\neg C$ to catch multiple declarations. Once C has been defined the rest of the Boolean grammar is quite readable; it completely specifies and checks all context conditions. Experiments show that the time complexity is about $O(n^2)$. A critical analysis closes the paper.

291. **Jackson**, Quinn Tyler. *Adapting to Babel: Adaptivity and Context-Sensitivity in Parsing*. In Press, 2006. The §-calculus (pronounced "meta-ess calculus") (Jackson [285]) is extended with a notation A-BNF, "Adaptive BNF", which is BNF extended with several grammar and set manipulation functions, including intersection with a set generated by a subgrammar. This allows full Turing power. A very simple example is a §-grammar (A-BNF) for palindromes: `S ::= $x('[a-zA-Z]') [S] x`; this means: to accept an `S`, accept one token from the input if it intersects with the set of letters and assign it to the variable `x`, optionally accept an `S`, and finally accept the token in variable `x`.
The implementation uses a pushdown automaton augmented with name-indexed tries (PDA-T) reminiscent of a nested stack automaton, and zillions of optimizations. The time complexity is unknown; in practice it is almost always less than $O(n^2)$ and always less than $O(n^3)$. Although §-grammars may be seen as generating devices, the author makes a strong point for seeing them as recognition devices.
All facets of the system are described extensively, with many examples.

18.2.7 Error Handling

292. **Aho**, A. V. and Peterson, T. G. A minimum-distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1(4):305–312, 1972. A CF grammar is extended with error productions so that it will produce Σ^*; this is effected by replacing each occurrence of a terminal in a rule by a non-terminal that produces said terminal "with 0 errors" and any amount of garbage, including ε, "with 1 or more errors". The items in an Earley parser are extended with a count, indicating how many errors were needed to create the item. An item with error count k is added only if no similar item with a lower error count is present already.

293. **Conway**, R. W. and Wilcox, T. R. Design and implementation of a diagnostic compiler for PL/I. *Commun. ACM*, 16(3):169–179, 1973. Describes a diagnostic PL/C compiler, using a systematic method for finding places where repair is required, but the repair strategy for each of these places is chosen by the implementor. The parser uses a separable transition diagram technique (see Conway [333]). The error messages detail the error found and the repair chosen.

294. **Lyon**, G. Syntax-directed least-errors analysis for context-free languages: a practical approach. *Commun. ACM*, 17(1):3–14, Jan. 1974. Discusses a least-error analyser, based on Earley's parser without look-ahead. The Earley items are extended with an error count, and the parser is started with items for the start of each rule, in each state set. Earley's scanner is extended

as follows: for all items with the dot in front of a terminal, the item is added to the same state set with an incremented error count and the dot after the terminal (this represents an insertion of the terminal); if the terminal is not equal to the input symbol associated with the state set, add the item to the next state set with an incremented error count and the dot after the terminal (this represents a replacement); add the item as it is to the next state set, with an incremented error count (this represents a deletion). The completer does its work as in the Earley parser, but also updates error counts. Items with the lowest error counts are processed first, and when a state set contains an item, the same item is only added if it has a lower error count.

295. **Graham**, Susan L. and Rhodes, Steven P. Practical syntactic error recovery. *Commun. ACM*, 18(11):639–650, Nov. 1975. See Section 16.5 for a discussion of this error recovery method.

296. **Horning**, James J. What the compiler should tell the user. In Friedrich L. Bauer and Jürgen Eickel, editors, *Compiler Construction, An Advanced Course, 2nd ed*, volume 21 of *Lecture Notes in Computer Science*, pages 525–548. Springer, 1976. Lots of good advice on the subject, in narrative form. Covers the entire process, from lexical to run-time errors, considering detection, reporting and possible correction. No implementation hints.

297. **Hartmann**, Alfred C. *A Concurrent Pascal Compiler for Minicomputers*, volume 50 of *Lecture Notes in Computer Science*. Springer, 1977. [Parsing / error recovery part only:] Each grammar rule is represented as a small graph; each graph is converted into a subroutine doing top-down recursive descent. To aid error recovery, a set of "key" tokens is passed on, consisting of the union of the FIRST sets (called "handles" in the text) of the symbols on the prediction stack, the intuition being that each of these tokens could, in principle, start a prediction if all the previous ones failed. This set is constructed and updated during parsing. Before predicting the alternative for a non-terminal *A*, all input tokens not in the key set at this place are skipped, if any. If that does not bring up a token from *A*'s FIRST set — and thus allow an alternative to be chosen — *A* is discarded and the next prediction is tried.

298. **Lewi**, J., Vlaminck, K. de, Huens, J., and Huybrechts, M. The ELL(1) parser generator and the error-recovery mechanism. *Acta Inform.*, 10:209–228, 1978. Presents a detailed recursive descent parser generation scheme for ELL(1) grammars, and also presents an error recovery method based on so-called *synchronization triplets* (*a*,*b*,*A*). *a* is a terminal from FIRST(*A*), *b* is a terminal from LAST(*A*). The parser operates either in parsing mode or in error mode. It starts in parsing mode, and proceeds until an error occurs. Then, in error mode, symbols are skipped until either an end marker *b* is found where *a* is the last encountered corresponding begin-marker, in which case parsing mode resumes, or a begin-marker *a* is found, in which case *A* is invoked in parsing mode. As soon as *A* is accepted, error-mode is resumed. The success of the method depends on careful selection of synchronization triplets.

299. **Mickunas**, M. Dennis and Modry, John A. Automatic error recovery for LR parsers. *Commun. ACM*, 21(6):459–465, June 1978. When an error is encountered, a set of provisional parsings of the beginning of the rest of the input (so-called *condensations*) are constructed: for each state a parsing is attempted and those that survive according to certain criteria are accepted. This yields a set of target states. Now the stack is "frayed" by partly or completely undoing any reduces; this yields a set of source states. Attempts are made to connect a source state to a target state by inserting or deleting tokens. Careful rules are given.

300. **Pennello**, Thomas J. and DeRemer, Frank L. A forward move algorithm for LR error recovery. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 241–254, Jan. 1978. Refer to Graham and Rhodes [295]. Backward moves are found to be detrimental to error recovery. The extent of the forward move is determined as follows. At the error, an LALR(1) parser is started in a state including *all* possible items. The thus extended automaton is run until it wants to reduce past the error detection point. The resulting right context is used in error correction. An algorithm for the construction of a reasonably sized extended LALR(1) table is given.

301. **Tanaka**, Eiichi and Fu, King-Sun. Error-correcting parsers for formal languages. *IEEE Trans. Comput.*, C-27(7):605–616, July 1978. Starts from a CF CYK parser based on a 2-form grammar. The entry for a recognized symbol A in the matrix contains 0, 1 or 2 pointers to its children, plus an error weight; the entry with the lowest error weight is retained. Next, the same error-correction mechanism is introduced in a context-sensitive CYK parser, for which see [1]. Full algorithms are given. Finally some theorems are proven concerning these parsers, the main one being that the error-correcting properties under these algorithms depend on the language only, not on the grammar used. High-threshold, notationally heavy paper, with extensive examples though.

302. **Fischer**, C. N., Tai, K.-C., and Milton, D. R. Immediate error detection in strong LL(1) parsers. *Inform. Process. Lett.*, 8(5):261–266, June 1979. A strong-LL(1) parser will sometimes perform some incorrect parsing actions, connected with ε-matches, when confronted with an erroneous input symbol, before signalling an error; this impedes subsequent error correction. A subset of the LL(1) grammars is defined, the *nullable LL(1) grammars*, in which rules can only produce ε directly, not indirectly. A special routine, called before an ε-match is done, hunts down the stack to see if the input symbol will be matched or predicted by something deeper on the stack; if not, an error is signaled immediately. An algorithm to convert any strong-LL(1) grammar into a non-nullable strong-LL(1) grammar is given. (See also Mauney and Fischer [309]).

303. **Fischer**, C. N., Milton, D. R., and Quiring, S. B. Efficient LL(1) error correction and recovery using only insertions. *Acta Inform.*, 13(2):141–154, 1980. See Section 16.6.4 for a discussion of this error recovery method.

304. **Pemberton**, Steven. Comments on an error-recovery scheme by Hartmann. *Softw. Pract. Exper.*, 10(3):231–240, 1980. Extension of Hartmann's error recovery scheme [297]. Error recovery in a recursive descent parser is done by passing to each parsing routine a set of "acceptable" symbols. Upon encountering an error, the parsing routine will insert any directly required terminals and then skip input until an acceptable symbol is found. Rules are given and refined on what should be in the acceptable set for certain constructs in the grammar.

305. **Röhrich**, Johannes. Methods for the automatic construction of error correcting parsers. *Acta Inform.*, 13(2):115–139, Feb. 1980. See Section 16.6.3 for a discussion of this error recovery method. The paper also discusses implementation of this method in LL(k) and LR(k) parsers, using so-called *deterministic continuable stack automata*.

306. **Anderson**, Stuart O. and Backhouse, Roland C. Locally least-cost error recovery in Earley's algorithm. *ACM Trans. Prog. Lang. Syst.*, 3(3):318–347, July 1981. Parsing and error recovery are unified so that error-free parsing is zero-cost error recovery. The information already present in the Earley items is utilized cleverly to determine possible continuations. From these and from the input, the locally least-cost error recovery can be computed, albeit at considerable expense. Detailed algorithms are given.

307. **Dwyer**, Barry. A user-friendly algorithm. *Commun. ACM*, 24(9):556–561, Sept. 1981. Skinner's theory of operant conditioning applied to man/machine interaction: tell the user not what is wrong but help him how to do better. In syntax errors this means showing what the parser understood and what the pertinent syntax rules are.

308. **Gonser**, Peter. *Behandlung syntaktischer Fehler unter Verwendung kurzer, fehlereinschließender Intervalle*. PhD thesis, Technical report, Technische Universität München, München, July 21 1981, (in German). Defines a syntax error as a minimal substring of the input that cannot be a substring of any correct input; if there are n such substrings, there are (at least) n errors. Finding such substrings is too expensive, but if we are doing simple precedence parsing and have a stack configuration $b \lessdot A_1 \odot A_2 \odot \cdots \odot A_k$ and a next input token c, where $\odot$ is either $\lessdot$ or $\doteq$ and there is no precedence relation between A_k and c, then the substring from which $bA_1A_2 \cdots A_kc$ was reduced must contain at least one error. The reason is that precedence information does not travel over terminals; only non-terminals can transmit information from left to right through the stack, by the choice of the non-terminal. So if the c cannot be understood, the cause cannot lie to the left of the b. This gives us an interval that is guaranteed to contain an error.

Several rules are given on how to turn the substring into an acceptable one; doing this successively for all error intervals turns the input into a syntactically correct one. Since hardly any grammar is simple precedence, several other precedence-like grammar forms are developed which are stronger and in the end cover the deterministic languages. See [130] for these.

309. **Mauney**, Jon and Fischer, Charles N. An improvement to immediate error detection in strong LL(1) parsers. *Inform. Process. Lett.*, 12(5):211–212, 1981. The technique of Fischer, Tai and Milton [302] is extended to all LL(1) grammars by having the special routine which is called before an ε-match is done do conversion to non-nullable on the fly. Linear time dependency is preserved by setting a flag when the test succeeds, clearing it when a symbol is matched and by not performing the test if the flag is set: this way the test will be done at most once for each symbol.

310. **Anderson**, S. O. and Backhouse, R. C. An alternative implementation of an insertion-only recovery technique. *Acta Inform.*, 18:289–298, 1982. Argues that the FMQ error corrector of Fischer, Milton and Quiring [303] does not have to compute a complete insertion. It is sufficient to compute the first symbol. If $w = w_1 w_2 \cdots w_n$ is an optimal insertion for the error a following prefix u, then $w_2 \cdots w_n$ is an optimal insertion for the error a following prefix uw_1. Also, immediate error detection is not necessary. Instead, the error corrector is called for every symbol, and returns an empty insertion if the symbol is correct.

311. **Anderson**, S. O., Backhouse, R. C., Bugge, E. H., and Stirling, C. P. An assessment of locally least-cost error recovery. *Computer J.*, 26(1):15–24, 1983. Locally least-cost error recovery consists of a mechanism for editing the next input symbol at least cost, where the cost of each edit operation is determined by the parser developer. The method is compared to Wirth's followset method (see Stirling [314]) and compares favorably.

312. **Brown**, P. J. Error messages: The neglected area of the man/machine interface?. *Commun. ACM*, 26(4):246–249, 1983. After showing some appalling examples of error messages, the author suggests several improvements: 1. the use of windows to display the program text, mark the error, and show the pertinent manual page; 2. the use of a syntax-directed editor to write the program; 3. have the parser suggest corrections, rather than just error messages. Unfortunately 1 and 3 seem to require information of a quality that parsers that produce appalling error messages just cannot provide.

313. **Richter**, Helmut. Noncorrecting syntax error recovery. *ACM Trans. Prog. Lang. Syst.*, 7(3):478–489, July 1985. Extends Gonser's method [308] by using suffix grammars and a reverse scan, which yields provable properties of the error interval. See Section 16.7 for a discussion of this method. Bounded-context grammars are conjectured to yield deterministic suffix grammars.

314. **Stirling**, Colin P. Follow set error recovery. *Softw. Pract. Exper.*, 15(3):239–257, March 1985. Describes the followset technique for error recovery: at all times there is a set of symbols that depends on the parse stack and that will not be skipped, called the *followset*. When an error occurs, symbols are skipped until one is found that is a member of this set. Then, symbols are inserted and/or the parser state is adapted until this symbol is legal. In fact there is a family of error recovery (correction) methods that differ in the way the followset is determined. The paper compares several of these methods.

315. **Choe**, Kwang-Moo and Chang, Chun-Hyon. Efficient computation of the locally least-cost insertion string for the LR error repair. *Inform. Process. Lett.*, 23(6):311–316, 1986. Refer to Anderson et al. [311] for locally least-cost error correction. The paper presents an efficient implementation in LR parsers, using a formalism described by Park, Choe and Chang [65].

316. **Kantorowitz**, E. and Laor, H. Automatic generation of useful syntax error messages. *Softw. Pract. Exper.*, 16(7):627–640, July 1986. Rules for useful syntax error messages: 1. Indicate a correction only if it is the only possibility. 2. Otherwise show the full list of legal tokens in the error position. 3. Mark skipped text.
To implement this the grammar is required to be LL(1) and each rule is represented internally by a syntax diagram. In case 1 the recovery is easy: perform the correction. Case 2 relies on an

"acceptable set", computed in two steps. First all paths in the present syntax diagram starting from the error point are searched for terminals that do not occur in the FIRST sets of non-terminals in the same syntax diagram. If that set is not empty it is the acceptable set. Otherwise the FOLLOW set is constructed by consulting the stack, and used as the acceptable set. Explicit algorithms given.

317. **Burke**, Michael G. and Fisher, Gerald A. A practical method for LL and LR syntactic error diagnosis and recovery. *ACM Trans. Prog. Lang. Syst.*, 9(2):164–197, April 1987. Traditional error recovery assumes that all tokens up to the error symbol are correct. The article investigates the option of allowing earlier tokens to be modified. To this end, parsing is done with two parsers, one of which is a number of tokens ahead of the other. The first parser does no actions and keeps enough administration to be rolled back, and the second performs the semantic actions; the first parser will modify the input stream or stack so that the second parser will never see an error. This device is combined with three error repair strategies: single token recovery, scope recovery and secondary recovery. In single token recovery, the parser is rolled back and single tokens are deleted, inserted or replaced by tokens specified by the parser writer. In scope recovery, closers as specified by the parser writer are inserted before the error symbol. In secondary recovery, sequences of tokens around the error symbol are discarded. In each case, a recovery is accepted if it allows the parser to advance a specified number of tokens beyond the error symbol. It is reported that this technique corrects three quarters of the normal errors in Pascal programs in the same way a knowledgeable human would. The effects of fine-tuning are discussed.

318. **Cormack**, Gordon V. An LR substring parser for noncorrecting syntax error recovery. *ACM SIGPLAN Notices*, 24(7):161–169, June 1989. Using the BC-SLR(1,1) substring parser from the same paper ([211]) the author gives examples of interval analysis on incorrect Pascal programs.

319. **Charles**, Philippe. An LR(k) error diagnosis and recovery method. In *Second International Workshop on Parsing Technologies*, pages 89–99, Feb. 1991. Massive approach to syntax error recovery, extending the work of Burke and Fisher [317], in four steps. 1. No information is lost in illegal reductions, as follows. During each reduction sequence, the reduce actions are stored temporarily, and actually applied only when a successful shift action follows. Otherwise the original stack is passed to the recovery module. 2. Primary (local) recovery: include merging the error token with its successor; deleting the error token; inserting an appropriate terminal in front of the error token; replacing the error token by a suitable terminal; inserting an appropriate non-terminal in front of the error token; replacing the error token by a suitable non-terminal. All this is controlled by weights, penalties and number of tokens that can be accepted after the modification. 3. Secondary (phrase-level) recovery: for a sequence of "important non-terminals" the unfinished phrase is removed from the stack and a synchronization is made, until a good one is found. Criteria for "important non-terminals" are given. 4. Scope recovery, in which nesting errors are repaired: For each self-embedding rule A, nesting information is precomputed, in the form of a scope prefix, a scope suffix, a look-ahead token, and a set of states. Upon error, these scopes are tested to bridge a possible gap over missing closing elements. The system provided excellent error recovery in a very large part of the cases tried.
Complete algorithms are given.

320. **Deudekom**, A. van and Kooiman, P. Top-down non-correcting error recovery in LLgen. Technical Report IR 338, Vrije Universiteit, Faculteit Wiskunde en Informatica, Amsterdam, Oct. 1993. Describes the addition of a Richter-style [313] error recovery mechanism to *LLgen*, an LL(1) parser generator, using a Generalized LL parser. The suffix grammar used by the mechanism is generated on the fly, and pitfalls concerning left recursion (a general problem in LL parsing), right recursion (a specific problem in error recovery), and ε-rules are pointed out and solved.
LLgen allows liberties with the LL(1) concept; these may interfere with automated error recovery. The conflict resolvers turned out to be no problem, but LLgen allows subparsers to be called from semantic actions, thus extending the accepted language, and syntax error messages to be given from semantic actions, thus restricting the accepted language. The error recovery grammar, however, has to represent the accepted language precisely; this necessitated two new parser generator directives.

Examples of error recovery and efficiency measurements are provided.
See also [170] for the Generalized LL parsing part.

321. **McKenzie**, Bruce J., Yeatman, Corey, and De Vere, Lorraine. Error repair in shift-reduce parsers. *ACM Trans. Prog. Lang. Syst.*, 17(4):672–689, July 1995. The two-stage technique described uses breadth-first search to obtain a series of feasible repairs, each of which is then validated. The first feasible validated repair is accepted.
To obtain feasible repairs, a priority queue of parser states each containing a stack, a representation of the rest of the input, a string of insert tokens, a string of deleted tokens and a cost is created in breadth-first fashion, ordered by cost. The top parser state in the queue is considered, a new state is created for each possible shift, with its implied inserted token, and a new state for the deletion of one token from the input, each of them with its cost. If one of these new states allows the parser to continue, it is deemed feasible and examined for validity.
The repair is valid if it allows the parser to accept the next N input tokens. If it is invalid, more parser states are created in the priority queue. If the queue gets exhausted, no error recovery is possible.
The paper contains much sound advice about implementing such a scheme. To reduce the number of parser states that have to be examined, a very effective pruning heuristic is given, which reduces the number by two or three orders of magnitude. In rare cases, however, the heuristic causes some cheaper repairs to be missed. See also Bertsch and Nederhof [323].

322. **Ruckert**, Martin. Generating efficient substring parsers for BRC grammars. Technical Report 98-105, State University of New York at New Paltz, New Paltz, NY 12561, July 1998. Error reporting and recovery using a BRC-based substring parser. For the parser see [217].

323. **Bertsch**, Eberhard and Nederhof, Mark-Jan. On failure of the pruning technique in "error repair in shift-reduce parsers". *ACM Trans. Prog. Lang. Syst.*, 21(1):1–10, Jan. 1999. The authors analyse the pruning heuristic presented in McKenzie et al. [321], and show that it can even cause the repair process to fail. A safe pruning heuristic is given, but it is so weak, and the failing cases are so rare, that the authors recommend to use the original but slightly faulty heuristic anyway.

324. **Ruckert**, Martin. Continuous grammars. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 303–310. ACM, 1999. Gives an example of a situation in which an error in the first token of the input can only be detected almost at the end of the input, invalidating almost all parsing done so far. To avoid such disasters, the author defines "continuous grammars", in which changing one token in the input can effect only limited changes in the parse tree: the mapping from string to parse tree is "continuous" rather than "discontinuous". This goal is achieved by imposing a metric for the distance between two nodes on a BCP grammar, and requiring that this distance is bounded by a constant for any single-token change in the input. It turns out that all bounded-context grammars are continuous; those bounded-context parsable are not, but can often be doctored.

325. **Cerecke**, Carl. Repairing syntax errors in LR-based parsers. *New Zealand J. Computing*, 8(3):3–13, June 2001. Improves McKenzie et al.'s algorithm [321] by limiting the lengths of circular search paths in the LR automaton. Left-recursive rules do not create cycles; right-recursive rules create cycles that have to be followed only once; and self-embedding rules create cycles that have to be followed until l symbols have been inserted, where l is the verification length. The improved parser solved 25% of the errors not solved by the original algorithm.

326. **Kim**, I.-S. and Choe, K.-M. Error repair with validation in LR-based parsing. *ACM Trans. Prog. Lang. Syst.*, 23(4):451–471, 2001. The combinations explored dynamically in McKenzie et al.'s algorithm [321] are computed statically during LR table generation, using a shortest-path search through the right-context graph.

327. **Corchuelo**, Rafael, Pérez, José A., Ruiz, Antonio, and Toro, Miguel. Repairing syntax errors in LR parsers. *ACM Trans. Prog. Lang. Syst.*, 24(6):698–710, Nov. 2002. The four LR parse action shift, reduce, accept, and reject are formalized as operators on a pair (stack, rest

of input). The error repair actions of an LR parser, insert, delete and forward move are described in the same formalism. ("Forward move" performs a limited number of LR parse actions, to see if there is another error ahead.)
The three error repair operators generate a search space, which is bounded by the depth of the forward move (N), the number of input tokens considered (N_t), and the maximum number of insertions (N_i) and deletions (N_d). The search space is searched breadth-first, with or without an error cost function; if the search space is found not to contain a solution, the system reverts to panic mode. The breadth-first search is implemented by a queue.
The system produces quite good but not superb error repair, is fast, and can easily be added to existing parsers, since it does not require additional tables and uses existing parsing actions only. With $N = 3$, $N_t = 10$, $N_i = 4$, and $N_d = 3$, the system almost always finds a solution; the solution is acceptable in about 85% of the cases. These results are compared to an extensive array of other error repair techniques.

328. **Jeffery**, Clinton L. Generating LR syntax error messages from examples. *ACM Trans. Prog. Lang. Syst.*, 25(5):631–640, Sept. 2003. The parser generator is provided with a list of error situations (pieces of incorrect code) with their desired error messages. The system then generates a provisional LR parser, runs it on each of the error situations, records in which state the parser ends up on which input token, and notes the triple (LR state, error token, error message) in a list. This list is then incorporated in the definitive parser, which will produce the proper error message belonging to the state and the input token, when it detects an error.

18.2.8 Incremental Parsing

329. **Lindstrom**, Gary. The design of parsers for incremental language processors. In *Second Annual ACM Symposium on Theory of Computing*, pages 81–91. ACM, 1970. The input is conceptually divided into "fragments" (substrings) by appointing by hand a set C of terminals that act as fragment terminators. Good candidates are separators like **`end`**, **`else`**, and **`;`**. Now for each non-terminal A in the grammar we create three new non-terminals: $<\!A$, which produces all prefixes of $\mathcal{L}(A)$ that end in a token in C, $A\!>$ for all suffixes, and $<\!A\!>$ for all infixes; rules for these are constructed. The input is then parsed by an LR parsing using these rules. The resulting fragments are saved and reused when the input is modified.
The parser does not know its starting state, and works essentially like the substring parser of Bates and Lavie [214], but the paper does not discuss the time complexity.

330. **Degano**, Pierpaolo, Mannucci, Stefano, and Mojana, Bruno. Efficient incremental LR parsing for syntax-directed editors. *ACM Trans. Prog. Lang. Syst.*, 10(3):345–373, July 1988. The non-terminals of a grammar are partitioned by hand into sets of "incrementally compatible" non-terminals, meaning that replacement of one non-terminal by an incrementally compatible one is considered a minor structural change. Like in Korenjak's method [53], for a partitioning in n sets $n+1$ parse tables are constructed, one for each set and one for the grammar that represents the connection between the sets. The parser user is allowed interactively to move or copy the string produced by a given non-terminal to a position where an incrementally compatible one is required. This approach keeps the text (i.e. the program text) reasonably correct most of the time and uses rather small tables.

331. **Vilares Ferro**, M. and Dion, B. A. Efficient incremental parsing for context-free languages. In *1994 International Conference on Computer Languages*, pages 241–252. IEEE Computer Society Press, May 1994. Suppose the GSS of a GLR parsing for a string w is available, and a substring $w_{i\cdots j}$ is replaced by a string u, possibly of different length. Two algorithms are supplied to update the GSS. In "total recovery" the smallest position $k \geq j$ is found such that all arcs (pops) from k reach back over i; the section $i \cdots k$ is then reparsed. Much technical detail is needed to make this work. "Partial recovery" preserves only those arcs that are completely to the right of the affected region. Extensive examples are given and many experimental results reported.

18.3 Parsers and Applications

18.3.1 Parser Writing

332. **Grau**, A. A. Recursive processes and ALGOL translation. *Commun. ACM*, 4(1):10–15, Jan. 1961. Describes the principles of a compiler for ALGOL 60, in which each entity in the language corresponds to a subroutine. Since ALGOL 60 is recursive in that blocks may contain blocks, etc., the compiler routines must be recursive (called "self-enslaving" in the paper); but the author has no compiler that supports recursive subroutines, so code segments for its implementation (routine entry, exit, stack manipulation, etc.) are provided. Which routine is called when is determined by the combination of the next input symbol and a state which is maintained by the parser. This suggests that the method is a variant of recursive ascent rather than of recursive descent.
The technique is demonstrated for a representative subset of ALGOL. In this demo version there are 13 states, determined by hand, and 17 token classes. The complete 13×17 matrix is provided; the contents of each entry is designed by considering exactly what must be done in that particular case.

333. **Conway**, Melvin E. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963. The first to introduce coroutines and to apply them to structure a compiler. The parser is Irons' [2], made deterministic by a No-Loop Condition and a No-Backup Condition. It follows transition diagrams rather than grammar rules.

334. **Tarjan**, R. E. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972. The power of depth-first search is demonstrated by two linear graph algorithms: a biconnectivity test and finding strongly connected components.
An undirected graph is *biconnected* if for any three nodes p, q, and r, you can go from p to q while avoiding r. The depth-first search on the undirected graph imposes a numbering on the nodes, which gives rise to beautiful palm trees.
A *strongly connected component* is a subset of the nodes of a directed graph such that for any three nodes p, q, and r in that subset, you can go from p to q while going through r.

335. **Aho**, A. V., Johnson, S. C., and Ullman, J. D. Deterministic parsing of ambiguous grammars. *Commun. ACM*, 18(8):441–452, 1975. Demonstrates how LL and LR parsers can be constructed for certain classes of ambiguous grammars, using simple disambiguating rules, such as operator-precedence.

336. **Glanville**, R. Steven and Graham, Susan L. A new method for compiler code generation (extended abstract). In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, 1978. SLR(1) parsing is used to structure the intermediate code instruction stream originating from a compiler front end. The templates of the target machine instructions form the grammar for the structuring; this grammar is almost always ambiguous and certainly not SLR(1). The parser actions are accompanied by actions that record semantic restrictions and costs. SLR(1) conflicts are resolved in 2 ways: upon shift/reduce conflicts the parser shifts; upon reduce/reduce conflicts the reduction with the longest reduce with the lowest cost which is compatible with the semantic restrictions is used. The parser cannot get stuck provided the grammar is "uniform". Conditions for a uniform grammar are given and full algorithms are supplied.

337. **Milton**, D. R., Kirchhoff, L. W., and Rowland, B. R. An ALL(1) compiler generator. *ACM SIGPLAN Notices*, 14(8):152–157, Aug. 1979. Presents an LL(1) parser generator and attribute evaluator which allows LL(1) conflicts to be solved by examining attribute values; the generated parsers use the error correction algorithm of Fischer, Milton and Quiring [303].

338. **Dencker**, Peter, Dürre, Karl, and Heuft, Johannes. Optimization of parser tables for portable compilers. *ACM Trans. Prog. Lang. Syst.*, 6(4):546–572, Oct. 1984. Given an $n \times m$ parser table, an $n \times m$ bit table is used to indicate which entries are error entries; this table is significantly smaller than the original table and the remaining table is now sparse (typically 90-98% don't-care entries). The remaining table is compressed row-wise (column-wise) by setting up

an interference graph in which each node corresponds to a row (column) and in which there is an edge between any two nodes the rows (columns) of which occupy an element in the same position. A (pseudo-)optimal partitioning is found by a minimal graph-coloring heuristic.

339. **Waite**, W. M. and Carter, L. R. The cost of a generated parser. *Softw. Pract. Exper.*, 15(3):221–237, 1985. Supports with measurements the common belief that compilers employing table-driven parsers suffer performance degradation with respect to hand-written recursive descent compilers. Reasons: interpretation of parse tables versus direct execution, attribute storage allocation and the mechanism to determine which action(s) to perform. Then, a parser interface is proposed that simplifies integration of the parser; implementation of this interface in assembly language results in generated parsers that cost the same as recursive descent ones. The paper does not consider generated recursive descent parsers.

340. **Aho**, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass., 1986. The "Red Dragon Book". Excellent, UNIX-oriented treatment of compiler construction. Even treatment of the various aspects.

341. **Cohen**, Jacques and Hickey, Timothy J. Parsing and compiling using Prolog. *ACM Trans. Prog. Lang. Syst.*, 9(2):125–164, April 1987. See same paper [26] for parsing techniques in Prolog. Shows that Prolog is an effective language to do grammar manipulation in: computation of FIRST and FOLLOW sets, etc.

342. **Koskimies**, Kai. Lazy recursive descent parsing for modular language implementation. *Softw. Pract. Exper.*, 20(8):749–772, Aug. 1990. Actually, it is lazy *predictive* recursive descent parsing for LL(1) grammars done such that each grammar rule translates into an independent module which knows nothing of the other rules. But prediction requires tables and tables are not modular. So the module for a rule A provides a routine STA(A) for creating at parse time the "start tree" of A; this is a tree with A at the top and the tokens in FIRST(A) as leaves (but of course FIRST(A) is unknown). STA(A) may call STA routines for other non-terminals to complete the tree, but in an LL(1) grammar this process will terminate; special actions are required if any of these non-terminals produces ε.
When during parsing A is predicted and a is the input token, a is looked up in the leaves of the start tree of A, and the path from that leaf to the top is used to expand A (and possibly its children) to produce A. This technique is in between non-predictive recursive descent and LL(1). Full code and several optimization are given.

343. **Norvig**, P. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, March 1991. Shows a general top-down parser in Common Lisp, which is based on a function which accepts a non-terminal N and a sequence of tokens I as inputs and produces a list of the suffixes of I that remain after prefixes that are produced by N have been removed. The resulting parser has exponential complexity, and the author shows that by memoizing the function (and some others) the normal $O(n^3)$ complexity can be achieved, supplying working examples. But the generation process loops on left-recursive grammar rules.

344. **Frost**, Richard A. Constructing programs as executable attribute grammars. *Computer J.*, 35(4):376–389, 1992. Introduces 4 combinators for parsing and processing of input described by an attribute grammar. Emphasis is on attribute evaluation rather than on parsing.

345. **Hutton**, Graham. Higher-order functions for parsing. *J. Functional Programming*, 2(3):323–343, 1992. By having the concatenation (invisible) and the alternation (vertical bar) from the standard grammar notation as higher-order functions, parsers can be written that are very close to the original grammar. Such higher-order functions — functions that take functions as parameters — are called combinators. The paper explains in detail how to define and use them, with many examples The resulting parser does breadth-first recursive descent CF parsing, provided the grammar is not left-recursive. The semantics of a recognized node is passed on as an additional parameter.
The ideas are then used to implement a simple pocket calculator language. The tiny system consists

of a layout analyser, a lexical analyser, a scanner, and a syntax analyser, each only a few lines long; these are then combined into a parser in one line. Methods to restrict the search are discussed.

346. **Leermakers**, René, Augusteijn, Lex, and Kruseman Aretz, Frans E. J. A functional LR parser. *Theoret. Comput. Sci.*, 104:313–323, 1992. An efficient formulation of an LR parser in the functional paradigm is given, with proof of correctness. It can do LR(0), LALR(1) and GLR.

347. **Rekers**, J. *Parser Generation for Interactive Environments*. PhD thesis, Technical report, Leiden University, Leiden, 1992. Discusses several aspects of incremental parser generation, GLR parsing, grammar modularity, substring parsing, and SDF. Algorithms in Lisp provided.

348. **Bod**, R. Using an annotated language corpus as a virtual stochastic grammar. In *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 778–783, Washington, DC, 1993. AAAI Press. A CF language is specified by a (large) set of annotated parse trees rather than by a CF grammar; this is realistic in many situations, including natural language learning. Probabilities are then derived from the set of trees, and parsing of new input strings is performed by weighted tree matching.

349. **Nederhof**, M.-J. and Sarbo, J. J. Efficient decoration of parse forests. In H. Trost, editor, *Feature Formalisms and Linguistic Ambiguity*, pages 53–78. Ellis Horwood, 1993. Concerns affix computation in AGFLs, of which the authors give a solid formal definition. Any CF method is used to obtain a parse forest. Each node in the forest gets a set of tuples, each tuple corresponding with one possible value set for its affixes. Expanding these sets of tuples would generate huge parse forests, so we keep the original parse forest and set up propagation equations. Sections ("cells") of the parse forest are isolated somewhat similar to basic blocks. Inside these cells, the equations are equalities; between the cells they are inclusions, somewhat similar to the dataflow equations between basic blocks. Additional user information may be needed to achieve uniqueness. Efficient implementations of the data structures are given.

350. **Frost**, R. A. Using memoization to achieve polynomial complexity of purely functional executable specifications of non-deterministic top-down parsers. *ACM SIGPLAN Notices*, 29(4):23–30, April 1994. The idea of obtaining a polynomial-time parser by memoizing a general one (see Norvig [343]) is combined with a technique to memoize functional-language functions, to obtain a polynomial-time parser in a functional language. A full example of the technique is given.

351. **Johnson**, Mark. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–418, 1995. Avoids the problem of non-termination of the creation of a list of suffixes in Norvig [343] by replacing the list by a function (a "continuation") which will produce the list when the times comes. Next the memoization is extended to the effect that a memo entry is prepared *before* the computation is made rather than after it. The author shows that in this setup left recursion is no longer a problem. Provides very clear code examples in Scheme.

352. **Kurapova**, E. W. and Ryabko, B. Y. Using formal grammars for source coding. *Problems of Information Transmission*, 31(1):28–32, 1995, (in Russian). The input to be compressed is parsed using a hand-written grammar and codes indicating the positions visited in the grammar are output; this stream is then compressed using Huffman coding. The process is reversed for decompression. Application to texts of known and unknown statistics is described, and the compression of a library of Basic programs using an LL(10) (!) character-level grammar is reported. The achieved results show a 10-30% improvement over existing systems. No explicit algorithms.

353. **Frost**, Richard A. and Szydlowski, Barbara. Memoizing purely functional top-down backtracking language processors. *Sci. Comput. Progr.*, 27(3):263–288, Nov. 1996. Using Hutton's combinators [345] yields a parser with exponential time requirements. This can be remedied by using memoization, bringing back the time requirement to the usual $O(n^3)$.

354. **Bhamidipaty**, A. and Proebsting, T. A. Very fast YACC-compatible parsers (for very little effort). *Softw. Pract. Exper.*, 28(2):181–190, 1998. Generate straightforward ANSI C code for each state of the LALR(1) parse table using **`switch`** statements, and let the C compiler worry over optimizations. The result is a *yacc*-compatible parser that is at most 30% larger, and about 4 times faster.

355. **Clark**, C. Build a tree — save a parse. *ACM SIGPLAN Notices*, 34(4):19–24, April 1999. Explains the difference between processing the nodes recognized during parsing on the fly and storing them as a tree. Obvious, but experience has shown that this has to be explained repeatedly.

356. **Sperber**, Michael and Thiemann, Peter. Generation of LR parsers by partial evaluation. *ACM Trans. Prog. Lang. Syst.*, 22(2):224–264, 2000. The techniques of Leermakers [155] are used to implement a recursive-ascent LR parser in Scheme. Constant propagation on the program text is then used to obtain a partial evaluation, yielding efficiencies that are comparable to those of *bison*.

357. **Metsker**, Steven John. *Building Parsers with Java*. Addison Wesley, 2001. Actually on how to implement "little languages" by using the toolkit package **`sjm.parse`**, supplied by the author. The terminology is quite different from that used in parsing circles. Grammars and non-terminals are hardly mentioned, but terminals are important. Each non-terminal corresponds to a parsing object, called a "parser", which is constructed from objects of class **`Repetition`**, **`Sequence`**, **`Alternation`** and **`Word`**; these classes (and many more) are supplied in the toolkit package. They represent the rule types $A \rightarrow B^*$, $A \rightarrow BC\cdots$, $A \rightarrow B|C|\cdots$, and $A \rightarrow t$, resp. Since each of these is implemented by calling the constructor of its components, B, C, ... cannot call A or a "parser class loop" would ensue; a special construction is required to avoid this problem (p. 105-106). But most little languages are not self-embedding anyway, except for the expression part, which is covered in depth.
The match method of a parser for A accepts a set of objects of class **`Assembly`**. An "assembly" contains a configuration (input string, position, and stack), plus an object representing the semantics of the part already processed. The match method of A produces another set of assemblies, those that appear after A has been matched and its semantics processed; the classes in the toolkit package just serve to lead these sets from one parser to the next. Assemblies that cannot be matched drop out; if there are FIRST/FIRST conflicts or FIRST/FOLLOW conflicts, assemblies are duplicated for each possibility. If at the end more than one assembly remains an error message is given; if none remains another error message is given. This implements top-down breadth-first parsing. It is interesting to see that this is an implementation of the 1962 "Multiple-Path Syntactic Analyzer" of Kuno and Oettinger [4].
The embedding in a programming language allows the match methods to have parameters, so very sophisticated context-sensitive matches can be programmed.
Chapters 1-9 explain how to build and test a parser; chapter 10 discusses some of the internal workings of the supplied classes; chapters 11-16 give detailed examples of implemented little languages, including a Prolog-like one, complete with unification; and chapter 17 gives further directions.
Tons of practical advice at a very manageable pace, allowing the user to quickly construct flexible parsers for little languages.

358. **Ljunglöf**, Peter. *Pure Functional Parsing: An Advanced Tutorial*. PhD thesis, Technical Report 6L, Chalmers University of Technology, Göteborg, April 2002. Consulted for its description of the Kilbury Chart Parser in Haskell. Assume the grammar to be in Chomsky Normal Form. Kilbury (chart) parsing proceeds from left to right, building up arcs marked with zero or more non-terminals A, which mean that A can produce the substring under the arc, and zero or more non-terminal pairs $B|C$, which mean that if this arc is connected on the right to an arc spanning C, both arcs together span a terminal production of B. For each token t, three actions are performed: Scan, Predict and Combine. Scan adds an arc spanning t, marked with all non-terminals that produce t. For each arc ending at and including t and marked A, Predict adds a mark $B|C$ to that arc for each rule $B \rightarrow AC$ in the grammar. For each arc ending at and including t, starting at position p, and marked A, Combine checks if there is an arc ending at p and marked $B|A$, and if so, adds an arc marked B, spanning both arcs.

The technique can be extended for arbitrary CF grammars. Basically, the markers are items, with the item $A \rightarrow \alpha\bullet\beta$ corresponding to the marker $A|\beta$.

359. **Sperberg-McQueen**, C. M. Applications of Brzozowski derivatives to XML schema processing. In *Extreme Markup Languages 2005*, page 26, Internet, 2005. IDEAlliance. Document descriptions in XML are based on "content models," which are very similar to regular expressions. It is important to find out if a content model C_1 "subsumes" a content model C_2, i.e., if there is a mapping such that the language of C_2 is included in the language of C_1. The paper shows how Brzozowski derivatives [138] can be used profitably for answering this and related questions.

18.3.2 Parser-Generating Systems

360. **Lesk**, M. E. and Schmidt, E. Lex: A Lexical Analyzer Generator. In *UNIX Manuals*, page 13. Bell Laboratories, Murray Hill, New Jersey, 1975. The regular grammar is specified as a list of regular expressions, each associated with a semantic action, which can access the segment of the input that matches the expression. Substantial look-ahead is performed if necessary. *lex* is a well-known and often-used lexical-analyser generator.

361. **Johnson**, Stephen C. YACC: Yet Another Compiler-Compiler. Technical report, Bell Laboratories, Murray Hill, New Jersey 07974, 1978. In spite of its title, *yacc* is one of the most widely used parser generators. It generates LALR(1) parsers from a grammar with embedded semantic actions and features a number of disambiguating and conflict-resolving mechanisms.

362. **Grune**, Dick and Jacobs, Ceriel J. H. A programmer-friendly LL(1) parser generator. *Softw. Pract. Exper.*, 18(1):29–38, Jan. 1988. Presents a practical ELL(1) parser generator, called *LLgen*, which generates fast error correcting recursive descent parsers. In addition to the error correction, *LLgen* features static as well as dynamic conflict resolvers and a separate compilation facility. The grammar can be viewed as a program, allowing for a natural positioning of semantic actions.

363. **Johnstone**, Adrian and Scott, Elizabeth. rdp: An iterator-based recursive descent parser generator with tree promotion operators. *ACM SIGPLAN Notices*, 33(9):87–94, Sept. 1998. Recursive descent parser generator with many add-ons: 1. A generalized BNF grammar structure `(` *expression* `)` *low* `@` *high separator*, which produces minimally *low* and maximally *high* productions of *expression*, separated by *separator*s. 2. Inlined extended ANSI-C code demarcated by `[*` and `*]`. 3. Inherited attributes as input parameters to grammar rules, and 1 synthetic attribute per grammar rule. This requires a rule to return two values: the Boolean success or failure value, and the synthetic attribute. An extended-code statement is provided for this. 4. Libraries for symbol tables, graph handling, scanning, etc. 5. Parse tree constructors, which allow the result of a subparse action to be attached to the parse tree in various places.
The parser is generalized recursive descent, for which see Johnstone and Scott [36].

18.3.3 Applications

364. **Kernighan**, B. W. and Cherry, L. L. A system for typesetting mathematics. *Commun. ACM*, 18(3):151–157, March 1975. A good example of the use of an ambiguous grammar to specify the preferred analysis of special cases.

365. **Share**, Michael. Resolving ambiguities in the parsing of translation grammars. *ACM SIGPLAN Notices*, 23(8):103–109, Aug. 1988. The UNIX LALR parser generator *yacc* is extended to accept LALR conflicts and to produce a parser that requests an interactive user decision when a conflict occurs while parsing. The system is used in document conversion.

366. **Evans**, William S. Compression via guided parsing. In *Data Compression Conference 1998*, pages 544–553. IEEE, 1998. To transmit text that conforms to a given grammar, the movements of the parser are sent rather than the text itself. For a top-down parser they are the rule numbers of the predicted rules; for bottom-up parsers they are the state transitions of the LR automaton. The packing problem is solved by adaptive arithmetic coding. The results are roughly 20% better than *gzip*.

367. **Evans**, William S. and Fraser, Christopher W. Bytecode compression via profiled grammar rewriting. *ACM SIGPLAN Notices*, 36(5):148–155, May 2001. The paper concerns the situation in which compressed bytecode is interpreted by on-the-fly decompression. The bytecode compression/decompression technique is based on the following observations.
1. Bytecode is repetitive and conforms to a grammar, so it can be represented advantageously as a parse tree in prefix form. Whenever the interpreter reaches a node representation, it knows the non-terminal (N) the node conforms to, exactly as with expressions in prefix form. The first byte of the node representation serves as a guiding byte and indicates which of the alternatives of the grammar rule N applies. This allows the interpreter again to know which non-terminal the next node conforms to, as required above.
2. Since non-terminals usually have few alternatives, most of the bits in the guiding bytes are wasted, and it would be better if all non-terminals had exactly 256 alternatives. One way to achieve this is to substitute some alternatives of some non-terminals in the alternatives of other non-terminals, thereby creating alternatives of alternatives, etc. This increases the number of alternatives per non-terminals and allows a more efficient representation of those subtrees of the parse tree that contain these alternatives of alternatives.
3. By choosing the substitutions so that the most frequent alternatives of alternatives are present in the grammar, a — heuristically — optimal compression can be achieved. The heuristic algorithm is simple: repeatedly substitute the most frequent non-terminal pair, unless the target non-terminal would get more than 256 alternatives in the process.
A few minor problems still have to be solved. The resulting grammar (expanded specifically for a given program) is ambiguous; an Earley parser is used to obtain the simplest — and most compact — parsing. Labels are dealt with as follows. All non-terminals that are ever a destination of a jump are made alternatives of the start non-terminal and parsing starts anew at each label. Special arrangements are made for linked-in code.
In one sample, the bytecode size was reduced from 199kB to 58kB, whereas the interpreter grew by 11kB, due to a larger grammar.

18.3.4 Parsing and Deduction

368. **Pereira**, Fernando C. N. and Warren, David H. D. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Mass., 1983. The Prolog deduction mechanism is top-down depth-first. It can be exploited to do parsing, using Definite Clause grammars. Parsing can be done more efficiently with Earley's technique. The corresponding Earley deduction mechanism is derived and analysed.

369. **Vilain**, Marc. Deduction as parsing: Tractable classification in the KL-ONE framework. In *National Conf. on Artificial Intelligence (AAAI-91), Vol. 1*, pages 464–470, 1991. The terms in the frame language KL-ONE are restricted as follows. The number of possible instances of each logic variable must be finite, and the free (existential) terms must obey a partial ordering. A tabular Earley parser is then sketched, which solves the "deductive recognition" in $O(\kappa^{3\alpha})$, where κ is the number of constants in ground rules, and α is the maximum number of terms in a rule.

370. **Rosenblueth**, David A. Chart parsers as inference systems for fixed-mode logic programs. *New Generation Computing*, 14(4):429–458, 1996. Careful reasoning shows that chart parsing can be used to implement fixed-mode logic programs, logic programs in which the parameters can be divided into synthesized and inherited ones, as in attribute grammars. Good explanation of chart parsers. See also Rosenblueth [371].

371. **Rosenblueth**, David A. and Peralta, Julio C. SLR inference: an inference system for fixed-mode logic programs based on SLR parsing. *J. Logic Programming*, 34(3):227–259, 1998. Uses parsing to implement a better Prolog. When a logic language clause is written in the form of a difference list $a(X_0,X_n)$:-$b_1(X_0,X_1),b_2(X_1,X_2),\ldots,b_n(X_{n-1},X_n)$, it can be related to a grammar rule $A \rightarrow B_1B_2\cdots B_n$, and SLR(1) techniques can be used to guide the search process. Detailed explanation of how to do this, with proofs. Lots of literature references. See also Rosenblueth [370].

372. **Vilares Ferro**, Manuel and Alonso Pardo, Miguel A. An LALR extension for DCGs in dynamic programming. In Carlos Martín Vide, editor, *Mathematical and Computational Analysis of Natural Language*, volume 45 of *Studies in Functional and Structural Linguistics*, pages 267–278. John Benjamins, 1998. First a PDA is implemented in a logic notation. Next a control structure based on dynamic programming is imposed on it, resulting in a DCG implementation. The context-free backbone of this DCG is isolated, and an LALR(1) table for it is constructed. This LALR(1) automaton is made to run simultaneously with the DCG interpreter, which it helps by pruning off paths. An explanation of the possible moves of the resulting machine is provided.

373. **Morawietz**, Frank. Chart parsing and constraint programming. In *18th International Conference on Computational Linguistics: COLING 2000*, pages 551–557, Internet, 2000. ACL. The straight-forward application of constraint programming to chart parsing has the inference rules of the latter as constraints. This results in a very obviously correct parser, but is inefficient. Specific constraints for specific grammars are discussed.

374. **Erk**, Katrin and Kruijff, Geert-Jan M. A constraint-programming approach to parsing with resource-sensitive categorial grammar. In *Natural Language Understanding and Logic Programming (NLULP'02)*, pages 69–86, Roskilde, Denmark, July 2002. Computer Science Department, Roskilde University. The parsing problem is reformulated as a set of constraints over a set of trees, and an existing constraint resolver is used to effectuate the parsing.

18.3.5 Parsing Issues in Natural Language Handling

375. **Yngve**, Victor H. A model and an hypothesis for language structure. *Proceedings of the American Philosophical Society*, 104(5):444–466, Oct. 1960. To accommodate discontinuous constituents in natural languages, the Chomsky CF grammar is extended and the language generation mechanism is modified as follows. 1. Rules can have the form $A \rightarrow \alpha\cdots\beta$, where the $\cdots$ is part of the notation. 2. Derivations are restricted to leftmost only. 3. A sentential form $\phi_1 \bullet AX\phi_2$, where $\bullet$ indicates the position of the derivation front, leads to $\phi_1 A \bullet \alpha X \beta \phi_2$; in other words, the right-hand side surrounds the next symbol in the sentential form. 4. The A in 3 remains in the sentential form, to the left of the dot, so the result is a derivation tree in prefix form rather than a sentence. 5. The length of the part of the sentential form after the dot is recorded in the derivation tree with each non-terminal; it is relevant since it represents the amount of information the speaker needs to remember in order to create the sentence, the "depth" of the sentence. Linguistic properties of this new device are examined.
The hypothesis is then that languages tend to use means to keep the depths of sentence to a minimum. Several linguistic phenomena are examined and found to support this hypothesis.

376. **Dewar**, Hamish P., Bratley, Paul, and Thorne, James P. A program for the syntactic analysis of English sentences. *Commun. ACM*, 12(8):476–479, 1969. The authors argue that the English language can be described by a regular grammar: most rules are regular already and the others describe concatenations of regular sublanguages. The finite-state parser used constructs the state subsets on the fly, to avoid large tables. Features (attributes) are used to check consistency and to weed out the state subsets.

377. **Chester**, Daniel. A parsing algorithm that extends phrases. *Am. J. Computational Linguistics*, 6(2):87–96, April 1980. A variant of a backtracking left-corner parser is described that is particularly convenient for handling continuing phrases like: "the cat that caught the rat that stole the cheese".

378. **Woods**, William A. Cascaded ATN grammars. *Am. J. Computational Linguistics*, 6(1):1–12, Jan. 1980. The grammar (of a natural language) is decomposed into a number of grammars, which are then *cascaded*; that is, the parser for grammar G_n obtains as input the linearized parse tree produced by the parser for G_{n-1}. Each grammar can then represent a linguistic hypothesis. Such a system is called an "Augmented Transition Network" (ATN). An efficient implementation is given.

379. **Shieber**, Stuart M. Direct parsing of ID/LP grammars. *Linguistics and Philosophy*, 7:135–154, 1984. In this very readable paper, the Earley parsing technique is extended in a straightforward way to ID/LP grammars (Gazdar et al. [381]). The items are still of the form $A \rightarrow \alpha \bullet \beta, i$, the main difference being that the β in an item is understood as the set of LP-acceptable permutations of the elements of the β in the grammar rule. Practical algorithms are given.

380. **Blank**, Glenn D. A new kind of finite-state automaton: Register vector grammar. In *Ninth International Conference on Artificial Intelligence*, pages 749–756. UCLA, Aug. 1985. In FS grammars, emphasis is on the states: for each state it is specified which tokens it accepts and to which new state each token leads. In *Register-Vector grammars* (*RV grammars*) emphasis is on the tokens: for each token it is specified which state it maps onto which new state(s). The mapping is done through a special kind of function, as follows. The state is a (global) vector (array) of registers (features, attributes). Each register can be *on* or *off*. For each token there is a condition vector with elements which can be *on*, *off* or *mask* (= *ignore*); if the condition matches the state, the token is allowed. For each token there is a result vector with elements which can be *on*, *off* or *mask* (= *copy*); if the token is applied, the result-vector elements specify how to construct the new state. ε-moves are incorporated by having tokens (called *labels*) which have ε for their representation. Termination has to be programmed as a separate register.
RV grammars are claimed to be compact and efficient for describing the FS component of natural languages. Examples are given. Embedding is handled by having a finite number of levels inside the state.

381. **Gazdar**, Gerald, Klein, Ewan, Pullum, Geoffrey, and Sag, Ivan. *Generalized Phrase Structure Grammar*. Basil Blackwell Publisher, Ltd., Oxford, UK, 1985. The phrase structure of natural languages is more easily and compactly described using *Generalized Phrase Structure Grammars* (*GPSGs*) or *Immediate Dominance/Linear Precedence grammars* than using conventional CF grammars. Theoretical foundations of these grammars are given and the results are used extensively in linguistic syntactic theory. GPSGs are not to be confused with general phrase structure grammars, aka Chomsky Type 0 grammars, which are called "unrestricted" phrase structure grammars in this book.
The difference between GPSGs, ID/LP grammars and CF grammars is explained clearly. A GPSG is a CF grammar, the non-terminals of which are not unstructured names but sets of *features* with their values; such compound non-terminals are called *categories*. An example of a feature is **`NOUN`**, which can have the values **`+`** or **`-`**; **`<NOUN,+>`** will be a constituent of the categories "noun phrase", "noun", "noun subject", etc.
ID/LP grammars differ from GPSGs in that the right-hand sides of production rules consist of multisets of categories rather than of ordered sequences. Thus, production rules (Immediate Dominance rules) define vertical order in the production tree only. Horizontal order in each node is restricted through (but not necessarily completely defined by) Linear Precedence rules. Each LP rule is considered to apply to every node; this is called the *Exhaustive Constant Partial Ordering property*.

382. **Blank**, Glenn D. A finite and real-time processor for natural language. *Commun. ACM*, 32(10):1174–1189, Oct. 1989. Several aspects of the register-vector grammars of Blank [380] are treated and extended: notation, center-embedding (3 levels), non-determinism through boundary-backtracking, efficient implementation.

383. **Abney**, Steven P. and Johnson, Mark. Memory requirements and local ambiguities of parsing strategies. *J. Psycholing. Res.*, 20(3):233–250, 1991. Based on the fact that parse stack space in the human brain is severely limited and that left-corner parsing requires exactly 2 stack entries for left-branching constructs and exactly 3 for right-branching, the authors conclude that neither top-down nor bottom-up parsing can be involved, but left-corner can.

384. **Resnik**, Philip. Left-corner parsing and psychological plausibility. In *14th International Conference on Computational Linguistics*, pages 191–197. Association for Computational Linguistics, 1992. Argues that the moment of composition of semantics is more important than the parsing technique; also in this respect a form of left-corner parsing is compatible with human language processing.

18.4 Support Material

18.4.1 Formal Languages

385. **Chomsky**, Noam. On certain formal properties of grammars. *Inform. Control*, 2:137–167, 1959. This article discusses what later became known as the Chomsky hierarchy. Chomsky defines type 1 grammars in the "context-sensitive" way. His motivation for this is that it permits the construction of a tree as a structural description. Type 2 grammars exclude ε-rules, so in Chomsky's system, type 2 grammars are a subset of type 1 grammars.
Next, the so called *counter languages* are discussed. A counter language is a language recognized by a finite automaton, extended with a finite number of counters, each of which can assume infinitely many values. $L_1 = \{a^n b^n | n > 0\}$ is a counter language, $L_2 = \{xy | x,y \in \{a,b\}^*,\ y$ is the mirror image of $x\}$ is not, so there are type 2 languages that are not counter languages. The reverse is not investigated.
The Chomsky Normal Form is introduced, but not under that name, and a bit different: Chomsky calls a type 2 grammar *regular* if production rules have the form $A \to a$ or $A \to BC$, with $B \neq C$, and if $A \to \alpha A \beta$ and $A \to \gamma A \eta$ then $\alpha = \gamma$ and $\beta = \eta$. A grammar is self-embedding if there is a derivation $A \stackrel{*}{\to} \alpha A \beta$ with $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$. The bulk of the paper is dedicated to the theorem that the extra power of type 2 grammars over type 3 grammars lies in this self-embedding property.

386. **Bar-Hillel**, Y., Perles, M., and Shamir, E. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961. (Reprinted in Y. Bar-Hillel, *Language and Information: Selected Essays on their Theory and Application*, Addison-Wesley, 1964, pp. 116-150.) Densely-packed paper on properties of context-free grammars, called *simple phrase structure grammars*, or SPGs here (this paper was written in 1961, two years after the introduction of the Chomsky hierarchy). All proofs are constructive, which makes the paper very important to implementers.
The main subjects are: any finite (one- and two-tape) automaton can be converted into a CF grammar; CF grammars are closed under reflection, union, product, and closure; CF grammars are not closed under intersection or complementation; almost any CF grammar can be made ε-free; almost any CF grammar can be made free of unit rules; it is decidable if a given CF grammar produces a given (sub)string; it is undecidable if the intersection of two CF grammars is a CF grammar; it is undecidable if the complement of a CF grammar is a CF grammar; it is undecidable if one CF grammar produces a sublanguage of another CF grammar; it is undecidable if one CF grammar produces the same language as another CF grammar; it is undecidable if a CF grammar produces a regular language; a non-self-embedding CF grammar produces a regular language; the intersection of a CF grammar and a FS automaton is a CF grammar.
Some of the "algorithms" described in this paper are impractical. For example, the decidability of parsing is proved by systematically producing all terminal productions up to the lengths of the input string, which is an exponential process. On the other hand, the intersection of a CF grammar and a

FS automaton is constructed in a time $O(n^d + 1)$, where n is the number of states in the automaton, and d is the maximum length of the RHSs in the grammar. This is the normal time complexity of general CF parsing. See also the same paper [219].

387. **Haines**, Leonard H. On free monoids partially ordered by embedding. *J. Combinatorial Theory*, 6:94–98, 1969. Proves that for any (infinite) set of words L (= subset of Σ^*) the following holds: 1. any language consisting of all subsequences of words in L is regular; 2. any language consisting of all words that contain subsequences of words in L is regular. This means that subsequence and supersequence parsing reduce to regular parsing.

388. **Cook**, Steven A. Linear time simulation of deterministic two-way pushdown automata. In *IFIP Congress (1)*, pages 75–80, 1971. Source of "Cook's Theorem": "Every 2-way deterministic pushdown automaton (2DPDA) language can be recognized in linear time on a random-access machine". A "proper arc" is a sequence of transitions of the 2DPDA for the given input that starts by pushing a stack symbol X, ends by popping the same X, and none of the in-between transitions pops the X. A "flat arc" is a single transition for the given input that neither pushes nor pops. Arcs are an efficient way to move the head over long distances without depending on or disturbing the stack underneath.
The algorithm starts by constructing all flat arcs, and from there builds all other arcs, until one connects the initial state to one of the final states. Since $|S|$ arcs can start at any point of the input, where $|S|$ is the number of transitions in the 2DPDA, and since each such arc has only one end point because the automaton is deterministic, there are only $|S|n$ arcs. The algorithm computes them so that no arc gets computed twice, so the algorithm is linear.
The theorem has many unexpected applications; see for example Aho's survey of algorithms for finding patterns in strings [147].

389. **Greibach**, Sheila A. The hardest context-free language. *SIAM J. Computing*, 2(4):304–310, Dec. 1973. The grammar is brought in Greibach Normal Form (Greibach [7]. Each rule $A \rightarrow aBCD$ is converted into a mapping $a \Rightarrow \bar{A}DCB$, which should be read as: "a can be replaced by a cancellation of prediction A, followed by the predictions D, C, and B, that is, in back-to-front order."
These mappings are used as follows. Suppose we have a grammar `S→aBC; B→b; C→c`, which yields the maps `a` $\Rightarrow$ `S̄CB`, `b` $\Rightarrow$ `B̄`, and `c` $\Rightarrow$ `C̄`. Now the input `abc` maps to `S̄CBB̄C̄`, which is prefixed with the initial prediction `S` to form `SS̄CBB̄C̄`. We see that when we view A and $\bar{A}$ as matching parentheses, we have obtained a well-balanced parenthesis string (wbps), and in fact the mapping of any correct input will be well balanced.
This makes parsing seem trivial, but in practice there will be more than one mapping for each terminal, and we have to chose the right one to get a wbps. The alternatives for each terminal are worked into the mapping by demarcating them with markers and separators, such that the mapping of any correct input maps to a conditionally well-balanced parenthesis string (cwbps), the condition being that the right segments are matched. These cwbpses form a CF language which depends on the symbols of the grammar only; the rules have been relegated to the mapping. (It is not shown that the cwbpses are a CF set.)
The dependency on the symbols of the grammar is removed by expressing them in unary notation: `B`, being the second non-terminal, is represented as `[xx[`, and `B̄` as `]x̄x̄]`, etc. With this representation, the cwbpses are not dependent on any grammar any more and any parsing problem can be transformed into them in linear time. So if we can parse cwbpses in time $O(n^x)$, we can parse any CF language in time $O(n^x)$, which makes cwbpses the *hardest context-free language*.

390. **Liu**, Leonard Y. and Weiner, Peter. An infinite hierarchy of intersections of context-free languages. *Math. Syst. Theory*, 7(2):185–192, May 1973. It is easy to see that the language $a^m b^n c^p \cdots a^m b^n c^p \cdots$ where there are k different $a, b, c \cdots$s, can be generated as the intersection of k CF languages: take for the first language $a^m b^* c^* \cdots a^m b^* c^* \cdots$, for the second language $a^* b^n c^* \cdots a^* b^n c^* \cdots$, etc. The authors then give a 6-page proof showing that the same cannot be achieved with $k-1$ languages; this proves the existence of the subject in the title.

391. **Hopcroft**, John E. and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachussetts, 1979. No-frills account of

formal language theory and computational (im)possibilities. Covers CYK and LR parsers, but as recognizers only.

392. **Heilbrunner**, Stephan. Tests for the LR-, LL-, and LC-regular conditions. *J. Comput. Syst. Sci.*, 27(1):1–13, 1983. Careful analysis shows that the LR-regular test in Čulik, II and Cohen's paper [57] is not correct. The repair leads to item grammars, which are right-regular grammars in which items are non-terminals. This mechanism is then used for very precise tests for LR-, LL-, and LC-regular-ness. Some proofs are given, but others are referred to a technical report.

393. **Rayward-Smith**, V. J. *A First Course in Formal Languages*. Blackwell Scientific, Oxford, 1983. Very useful intermediate between Révész [394] and Hopcroft and Ullman [391]. Quite readable (the subject permitting); simple examples; broad coverage. No treatment of LALR, no bibliography.

394. **Révész**, György E. *Introduction to Formal Languages*. McGraw-Hill, Singapore, 1985. This nifty little book contains many results and elementary proofs of formal languages, without being "difficult". It gives a description of the ins and outs of the Chomsky hierarchy, automata, decidability and complexity of context-free language recognition, including the hardest context-free language. Parsing is discussed, with descriptions of the Earley, LL(k) and LR(k) algorithms, each in a few pages.

395. **Geffert**, Viliam. A representation of recursively enumerable languages by two homomorphisms and a quotient. *Theoret. Comput. Sci.*, 62:235–249, 1988. Imagine the following mechanism to generate strings. The mechanism uses two homomorphisms h_1 and h_2 (a *homomorphism* is a translation table from tokens to strings of zero or more tokens) and an alphabet Σ; the tokens in the translation tables may or may not be in Σ. Now take an arbitrary string α, and construct the two translations $h_1(\alpha)$ and $h_2(\alpha)$. If it now so happens that $h_2(\alpha) = h_1(\alpha)w$ (so $h_1(\alpha)$ is the head of $h_2(\alpha)$ and w is the tail), and w consists of tokens that all happen to be in the alphabet Σ, then we keep w; otherwise α leads nowhere.
The author shows that this mechanism is equivalent to a Chomsky Type 0 grammar, and that the grammar defines the two homomorphisms and vice versa. The details are complicated, but basically h_1 and h_2 are such that as α grows, h_2 grows faster than h_1. The consequence is that if we want to extend α by a few tokens δ, the translation of δ through h_1 must match tokens already produced long ago by $h_2(\alpha)$ or α will be rejected; so very soon our hand is forced. This effect is used to enforce the long-range relationships characteristic of general phrase-structure grammars. In fact, α is some encoding of the derivation of w.

396. **Billington**, David. Using the context-free pumping lemma. *Commun. ACM*, 36(4):21, 81, April 1993. Short note showing a somewhat sharper lemma, better suited for proving that a language is not CF.

397. **Sudkamp**, Thomas A. *Languages and Machines*. Addison-Wesley, second edition, 1997. Carefully reasoned, very readable, but sometimes dull introduction to formal languages, with serious attention to grammars and parsing. FSA minimization, grammar manipulation, proof techniques for the equivalence of a grammar and a language, same for non-existence of a grammar for a language, etc. Many fully worked out examples.
Consists of five parts: CF grammars and parsing; automata and languages; decidability and computation; computational complexity; deterministic parsing.

398. **Schmitz**, Sylvain. Conservative ambiguity detection in context-free grammars. Technical Report RR-2006-30-FR, Université de Nice, Nice, 2006. A grammar G is represented in a way similar to Figure 9.48. On the basis of this representation an infinite graph is defined in which each node represents a rightmost sentential form of G. G is ambiguous of there is more than one path from a given node to another node in this graph. The infinite graph is rendered finite by defining equivalence relations between nodes that preserve the multiple paths if they exist. Testing the finite graph for multiple paths is simple. A lattice of possible equivalence relations is presented. The time complexity is $O(|G|^2|T|^{4k})$, where $|G|$ is the size of the grammar, $|T|$ is the number of terminals, and k depends on the equivalence relation.

18.4.2 Approximation Techniques

399. **Pereira**, Fernando C. N. and Wright, Rebecca N. Finite-state approximation of phrase-structure grammars. In *29th Annual Meeting of the Association for Computational Linguistics*, pages 246–255. Association for Computational Linguistics, 1991. The idea is to "flatten" the LR(0) automaton of a grammar G into an FSA that will accept any string from G, and not very much more. But the LR(0) automaton stops at reduce states, whereas the FSA has to continue. For any state s which contains an item $A \to \alpha\bullet$, all paths are searched backwards to states t where the item started. (Cf. the **lookback** relation from Section 9.7.1.3.) Each t_i has a transition on A to a state u_i. Now ε-transitions are added from s to each u_i.
This is the version that keeps no stack at all. It can be improved by keeping finite simplifications of the stack, and several variants are examined in great detail and with full theoretical support. For all left-linear and right-linear grammars and some CF grammars the approximation is exact.

400. **Pereira**, Fernando C. N. and Wright, Rebecca N. Finite-state approximation of phrase-structure grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 149–173. MIT Press, 1997. The LR(0) automaton of a grammar is "flattened" by ignoring the stack and replacing any reduction to A by an ε-transition to all states with incoming arrows marked A. This yields too coarse automata, even for regular and finite grammars. Rather than ignoring the stack, stack configurations are simulated and truncated as soon as they begin to repeat. This yields unwieldy automata. To remedy this the grammar is first decomposed into subgrammars by isolating strongly connected components in the grammar graph. Full algorithms and proofs are given. Sometimes the grammar needs to be modified (left-factored) to avoid exponential blow-up. See also [399, 404].

401. **Nederhof**, Mark-Jan. Regular approximations of CFLs: A grammatical view. In H. Bunt and A. Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, pages 221–241. Kluwer Academic Publishers, 2000. A regular envelope of a CF grammar is constructed by finding the self-embedding rules in it and splitting them in a left-recursive and a right-recursive persona. Many other regular approximating algorithms are discussed and compared.

402. **Nederhof**, Mark-Jan. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44, 2000. A regular envelope of a CF grammar is constructed by assigning a start state and a stop state to each non-terminal and $m+1$ intermediate states to each rule $A \to X_1 \cdots X_m$. These states are then connected by transitions, to form a transition network for the entire grammar. Properties of this approximation are investigated, and the algorithm is refined. It is compared empirically to other algorithms, where it proves effective, especially for large grammars.

403. **Yli-Jyrä**, Anssi. Regular approximations through labeled bracketing. In *Formal Grammar 2003*, pages 189–201. European Summer School in Logic Language and Information, 2003. A CF language consists of a nesting component, described by a grammar for nesting brackets (i.e. the sections of text that are forced to nest), and a regular component, which describes the shapes of these brackets. The nesting part can be decomposed in separate grammars for each nesting set of brackets. By judiciously restricting the various components, good and compact approximations to CF languages can be obtained. Properties of the various possibilities are examined.

404. **Pereira**, Fernando C. N. and Wright, Rebecca N. Finite-state approximation of phrase-structure grammars. Technical report, AT&T Reseach, Murray Hill, NJ, March 2005. Revised and extended version of Pereira and Wright [399, 400].

18.4.3 Transformations on Grammars

405. **Foster**, J. M. A syntax-improving program. *Computer J.*, 11(1):31–34, May 1968. The parser generator SID (Syntax Improving Device) attempts to remove LL(1) conflicts by eliminat-

ing left recursion, and then left-factoring, combined with inline substitution. If this succeeds, SID generates a parser in machine language.

406. **Hammer**, Michael. A new grammatical transformation into LL(k) form. In *Sixth Annual ACM Symposium on Theory of Computing*, pages 266–275, 1974. First an LR(k) automaton is constructed for the grammar. For each state that predicts only one non-terminal, say A, a new LR(k) automaton is constructed with A as start symbol, etc. This process splits up the automaton into many smaller ones, each using a separate stack, hence the name "multi-stack machine". For all LL(k) grammars this multi-stack machine is cycle-free, and for many others it can be made so, using some heuristics. In that case the multi-stack machine can be converted to an LL(k) grammar. This works for all LC(k) grammar and more. An algorithm for repairing the damage to the parse tree is given. No examples.

407. **Mickunas**, M. D., Lancaster, R. L., and Schneider, V. B. Transforming LR(k) grammars to LR(1), SLR(1) and (1,1) bounded right-context grammars. *J. ACM*, 23(3):511–533, July 1976. The required look-ahead of k tokens is reduced to $k-1$ by incorporating the first token of the look-ahead into the non-terminal; this requires considerable care. The process can be repeated until $k=1$ for all LR(k) grammars and even until $k=0$ for some grammars.

408. **Rosenkrantz**, D. J. and Hunt, H. B. Efficient algorithms for automatic construction and compactification of parsing grammars. *ACM Trans. Prog. Lang. Syst.*, 9(4):543–566, Oct. 1987. Many grammar types are defined by the absence of certain conflicts: LL(1), LR(1), operator-precedence, etc. A simple algorithm is given to modify a given grammar to avoid such conflicts. Modification is restricted to the merging of non-terminals and possibly the merging of terminals; semantic ambiguity thus introduced will have to be cleared up by later inspection. Proofs of correctness and applicability of the algorithm are given. The maximal merging of terminals while avoiding conflicts is also used to reduce grammar size.

18.4.4 Miscellaneous Literature

This section contains support material that is not directly concerned with parsers or formal languages.

409. **Warshall**, Stephen. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962. Describes how to obtain B^*, where B is an $n \times n$ Boolean matrix, in $O(n^3)$ actions, using a very simple 3-level loop.

410. **Michie**, D. "memo" functions and machine learning. *Nature*, 218(5136):19–22, April 6 1968. Recognizes that a computer function should behave like a mathematical function: the input determines the output, and how the calculation is done is immaterial, or at least behind the screens. This idea frees the way for alternative implementations of a given function, in this case by memoization.
A function implementation consists of a rote part and a rule part. The rote part contains the input-to-output mappings the function has already learned by rote, and the rule provides the answer if the input is new. New results are added to the rote part and the data is ordered in order of decreasing frequency by using a self-organizing list. This way the function "learns" answers by rote as it is being used. The list is fixed-size and if it overflows, the least popular element is discarded. Several examples of applications are given.

411. **Bhate**, Saroja and Kak, Subhash. Pāṇini's grammar and computer science. *Annals of the Bhandarkar Oriental Research Institute*, 72:79–94, 1993. In the Aṣṭādhyāyī, the Sanskrit scholar Pāṇini (probably c. 520-460 B.C.) gives a complete account of the Sanskrit morphology in 3,959 rules. The rules are context-free substitution rules with simple context conditions, and can be interpreted mechanically. The basic form is $A \rightarrow B(C)$, which means that A must be replaced by B if condition C applies; in the Sanskrit text, the separators $\rightarrow$, (and) are expressed through case endings. Macros are defined for many ordered sets. For example, the rule *iko yaṇ aci* means: *i*, *u*,

ṛ, and *ḷ* must be replaced by *y*, *v*, *r* and *l* respectively, when a vowel follows. All three words are macros: *ikaḥ* stands for the ordered set *iuṛḷ*; *yaṇ* stands for *yvrl*; and *ac* stands for all vowels. The rules literally means "of-*ikaḥ* [must come] *yaṇ* at-*ac*".
The rules differ from that of a CF grammar: 1. they have context conditions; 2. replacement is from a member of an ordered set to the corresponding member of the other ordered set; 3. the rules are applied in the order they appear in the grammar; 4. rule application is obligatory. Because of this Pāṇini's rules are more similar to a Unix *sed* script.
The authors explain several other features, all in computer science terms, and consider further implications for computer science and linguistics.

412. **Nuutila**, Esko. An efficient transitive closure algorithm for cyclic digraphs. *Inform. Process. Lett.*, 52(4):207–213, Nov. 1994. Very careful redesign of the top-down transitive closure algorithm using strongly connected components, named COMP_TC. Extensive experimental analysis, depicted in 3D graphs.

413. **Thompson**, Simon. *Haskell: The Craft of Functional Programming*. Addison Wesley, Harlow, England, 2nd edition, March 1999. Functional programming in Haskell and how to apply it. Section 17.5 describes a straightforward top-down parser; it is formulated as a monad on page 405.

414. **Grune**, Dick, Bal, Henri E., Jacobs, Ceriel J. H., and Langendoen, Koen G. *Modern Compiler Design*. John Wiley, Chichester, UK, 2000. Describes, among other things, LL(1), LR(0) and LR(1) parsers and attribute grammar evaluators in a compiler-design setting.

415. **Cormen**, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. Extensive treatment of very many subjects in algorithms, breadth-first search, depth-first search, dynamic programming, on which it contains a large section, topological sort, etc., etc.

416. **Goodrich**, Michael T. and Tamassia, Roberto. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley and Sons, 2nd edition, 2002. Low-threshold but extensive and in-depth coverage of algorithms and their efficiency, including search techniques, dynamic programming, topological sort.

417. **Sedgewick**, Robert. *Algorithms in C/C++/Java: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley, Reading, Mass., 2001/2002. Comprehensive, understandable treatment of many algorithms, beautifully done. Available for C, C++ and Java.

A

Hints and Solutions to Selected Problems

Answer to Problem 2.1: The description is not really finite: it depends on all descriptions in the list, of which there are infinitely many.

Answer to Problem 2.2: Any function that maps the integers onto unique integers will do (an *injective function*), for example n^n or the n-th prime.

Answer to Problem 2.3: Hint: use a special (non-terminal) marker for each block the turtle is located to the east of its starting point.

Answer to Problem 2.4: The terminal production cannot contain `S`, and the only way to get rid of it is to apply `S`$\rightarrow$`aSQ` $k \geq 0$ times followed by `S`$\rightarrow$`abc`. This yields a sentential form $\texttt{a}^k\texttt{abcQ}^k$. The part $\texttt{bcQ}^k$ cannot produce any `a`s, so all the `a`s in the result come from $\texttt{a}^k\texttt{a}$, so all strings $\texttt{a}^n\cdots$ can be produced, for $n = k+1 \geq 1$. We now need to prove that $\texttt{bcQ}^k$ produces exactly $\texttt{b}^n\texttt{c}^n$. The only way to get rid of the `Q` is through `bQc`$\rightarrow$`bbcc`, which requires the `Q` to be adjacent to the `b`. The only way to get it there is through repeated application of `cQ`$\rightarrow$`Qc`. This does not affect the number of each token. After j application of `bQc`$\rightarrow$`bbcc` we have $\texttt{bb}^j\texttt{c}^j\texttt{cQ}^{k-j}$, and after k applications we have $\texttt{bb}^k\texttt{c}^k\texttt{c}$, which equals $\texttt{b}^n\texttt{c}^n$.

Answer to Problem 2.5: Assume the alphabet is {`(`, `[`}. First create a nesting string of `(`, `)`, `[`, and `]`, with a marker in the center and another marker at the end: `([[...[([X])]...]])Z`. Give the center marker the ability to "ignite" the `)` or `]` on its right: `([[...[([X]`*`)]...]])Z`. Make the ignited parenthesis move right until it bumps into the end marker: `([[...[([X)]...]])]`*`Z`. Move it over the end marker and turn it into its (normal) partner: `([[...[([X)]...]])]Z[`. This moves the character in the first position after the center marker to the first position after the end marker; repeat for the next character: `([[...[([X]...]])]Z([`. This reverses the string between the markers while at the same time translating it: `([[...[([XZ([[...[([`. Now the markers can annihilate each other.

Answer to Problem 2.11: We have to carve up $\texttt{a}^i\texttt{b}^i$ into u, v, and w, such that uv^nw is again in $\texttt{a}^i\texttt{b}^i$, for all n. The problem is to find v: 1. v cannot lie entirely in $\texttt{a}^i$, since uv^nw would produce $\texttt{a}^{i+n}\texttt{b}^i$. 2. v cannot straddle the `a`-`b` boundary in $\texttt{a}^i\texttt{b}^i$, since even vv would contain a `b` followed by an `a`. 3. v cannot lie entirely in $\texttt{b}^i$either, for the same reason as under 1.

Answer to Problem 2.12: It depends. In $\mathtt{S_s \rightarrow A}$; $\mathtt{S \rightarrow B}$; $\mathtt{A \rightarrow a}$; $\mathtt{B \rightarrow a}$ the rules involving `B` can be removed without changing the language produced, but if one does so, the grammar is suddenly no longer ambiguous. Is that what we want?

Answer to Problem 3.1: Not necessarily. Although there is no ambiguity in the provenance of each terminal in the input, in that each can come from only one rule identifying one non-terminal, the composition of the non-terminals into the start symbol can still be ambiguous: $\mathtt{S \rightarrow A \mid B}$; $\mathtt{A \rightarrow C}$; $\mathtt{B \rightarrow C}$; $\mathtt{C \rightarrow a}$. See also Problem 7.3.

Answer to Problem 3.3: In a deterministic top-down parser the internal administration resides in the stack and the partial parse tree in the calls to semantic routines.

Answer to Problem 3.5: $\mathtt{S_s \rightarrow aS \mid b}$ with the input $\mathtt{a}^{k-1}\mathtt{b}$.

Answer to Problem 3.8: See Tomita [161].

Answer to Problem 3.12: 1b. Compute how many t_2s on the left and how many t_1s on the right each right-hand side contributes, using matching in between, and see if it comes out to 0 and 0 for the start symbol. 2. Assume the opposite. Then there is a (u_1, u_2)-demarcated string U which is not balanced for (t_1, t_2). Suppose U contains an unmatched t_1. Since the entire string is balanced for (t_1, t_2), the corresponding t_2 must be somewhere outside U. Together with the t_1 it forms a string T which must be balanced for (u_1, u_2). However, T does either not contain the u_1 or the u_2 boundary token of U, which means that it contains either an unbalanced u_1 or an unbalanced u_2. 3. Exhaustive search seems the only option. 4. Trivial, but potentially useful.

Answer to Problem 4.2: A simple solution can be obtained by replacing each terminal t_i in the original grammar by a non-terminal T_i, and add a rule $T_i \rightarrow t_1 | \varepsilon$ to the grammar. Now the standard Unger and CYK parser will recognize subsequences. Incorporating the recovery of the deleted tokens in the parser itself is much more difficult.

Answer to Problem 4.3: What does your algorithm do on $\mathtt{S \rightarrow AAA}$, $\mathtt{A \rightarrow}\varepsilon$ (multiple nullable right-hand side symbols)? On $\mathtt{S \rightarrow AS \mid x}$, $\mathtt{A \rightarrow AS \mid B}$, $\mathtt{B \rightarrow}\varepsilon$ (infinite ambiguity)?

Answer to Problem 4.4: No rule and no non-terminal remains, which supports the claim in Section 2.6 that ({}, {}, {}, {}) is the proper representation for grammars which produce the empty set.

Answer to Problem 4.7: For each rule of the form $A \rightarrow BC$, for each length l in $T_{i,B}$, for each length m in $T_{i+l,C}$, $T_{i,A}$ has to contain a length $l+m$.

Answer to Problem 5.2: 1. Yes: create a new accepting state $\Diamond$ and add ε-transitions to it from all original accepting states. 2. Yes: split off new transitions to $\Diamond$ just before you enter an accepting state. 3. No: an ε-free automaton deterministically recognizing just `a` and `ab` must have at least two different accepting states.

Answer to Problem 5.3: It is sufficient to show that removing non-productive rules from a regular grammar with no unreachable non-terminals will not result in a grammar with unreachable non-terminals. Now, suppose that a non-terminal U becomes unreachable when removing non-productive rules. This means that U occurs in some right-hand side that is removed because it is non-productive. But, as the grammar is regular, U is the only non-terminal occurring in this right-hand side, which means that U is non-productive and will be removed.

Answer to Problem 5.4: 1. Find all ε-loops and combine all nodes on each of them into a single node. 2. As long as there is an ε-transition $S_i \xrightarrow{\varepsilon} S_j$ do: for each transition $S_h \xrightarrow{a} S_i$ add a transition $S_h \xrightarrow{a} S_j$, for a terminal or ε. Then remove the transition $S_i \xrightarrow{\varepsilon} S_j$.

Answer to Problem 6.4: 1. Define a stack of procedure pointers *S*. In the procedure for $A \rightarrow BCD$, stack *CD*, call *B*, and unstack. In the procedure for $A \rightarrow a$, pop the top-of-stack procedure, call it if the tokens match, and push it back. The stack represents the prediction stack. 2. Define a record *L* as a link in a chain of pointers to procedures. In the procedure for $A \rightarrow BCD$, which has one parameter, a pointer `prev` to *L*, declare an *L* `this`, link it to `prev`, insert a pointer to *CD*, and call *B* with a pointer to `this` as parameter. In the procedure for $A \rightarrow a$, call the procedure in `prev` with the backlink in it as a parameter, if the tokens match. The linked list represents the prediction stack.

Answer to Problem 6.6: Hint: mind the ε-rules.

Answer to Problem 6.8: (*a*) See Nederhof [105].

Answer to Problem 7.2: The plan does not tell how many εs should be recognized at a given position and in which order. In particular, for infinitely ambiguous grammars infinitely many εs would have to be recognized.

Answer to Problem 7.3: No. Any grammar can be trivially brought in that form by introducing non-terminals for all terminals. See also Problem 3.1.

Answer to Problem 7.5: Lazy evaluation (+ memoization) is useful only when there is a good chance that the answer will not be needed. Since the Completer will come and visit anyway, it is not useful here. The required data structures would only slow the parser down.

Answer to Problem 7.6: Put edges for all rules from each node to itself in the chart. This fulfills the invariant. Put edges for the input tokens on the stack. The parser now makes roughly the same movements as a CYK parser.

Answer to Problem 7.7: Essentially the same arguments as in Section 7.3.6.

Answer to Problem 8.1: Each rule has only one alternative, so no decision is required, and the language produced contains exactly one string.

Answer to Problem 8.2: No. The first step in converting to CNF is ε-rule removal, which in general will not leave an LL(1) grammar. For example: **A→a|ε**, **B→cA** results in **B→cA′|c**, **A′→a**, which is not LL(1).

Answer to Problem 8.3: Yes, these substitutions do not change the FIRST and FOLLOW sets of the right-hand sides.

Answer to Problem 8.4: The token *t* is not in any FIRST set and either the grammar has no rules producing ε or *t* is not in any FOLLOW set either. (Question: How can a token not be in any FIRST nor in any FOLLOW set?)

Answer to Problem 8.5: *a*. Yes. The terminal productions of both alternatives of **S** start with different terminals in spite of the fact that the alternatives themselves start with the same non-terminal, the latter producing only ε. *b*. No. Now **A** can also produce **a**, which allows both alternatives to start with **a**, thus causing a FIRST/FIRST conflict. *c*. It depends on your precise definition. Pro: 1. There are no conflicts. 2. A naively generated parser would properly reject all finite strings, and would loop on $\mathbf{a}^{\infty}$. Con: The grammar is not practical, and it seems reasonable to require the grammar to be cleaned; but even Aho and Ullman [152, pg. 336] do not explicitly make this requirement.

Answer to Problem 8.6: To choose between the alternatives which produce ε, we need to include the FOLLOW sets. Both alternatives have FOLLOW(**S**), so we get a (rare) FOLLOW/FOLLOW conflict.

Answer to Problem 9.3: If the grammar happens to be LR(1) your parser will be relatively OK; it will just give late error messages. If the grammar happens not to be LR(1) (for example Figure 9.13) your parser may reject correct input.

Answer to Problem 9.8: Not necessarily, since the converted grammar may require earlier identification of handles.

Answer to Problem 9.9: Yes; substitution can only result in later identification of handles.

Answer to Problem 9.10: Yes. If in an item $A \rightarrow B \bullet C \ \square$ the non-terminal C produces ε, its channels will propagate $\square$ and spontaneously generate FIRST(C) to the station $\bullet C \ \square$.

Answer to Problem 9.12: See Charles [88, page 28].

Answer to Problem 9.17: `P_1→ε; P_2→T|P_6T; P_3→n|P_6n|E-n|P_6E-n; P_4→E; P_5→E$; P_6→[(|E-()[(|E-()*; P_7→E-|P_6E-; P_8→E-T|P_6E-T; P_9→P_6E; P_10→P_6E)`

Answer to Problem 9.20: For a parsing to be successful, the right context must match in number that which is on the stack; more in particular, the item `S→a•` is applicable only when there are just as many `a`s on the stack as there are in the rest of the input. The regular expressions cannot enforce this.

Answer to Problem 9.21: See Fortes Gálvez [95, Figure 6.3]

Answer to Problem 9.22: Since the input is not in the right context of the item, and since the regular envelopes are disjunct, it is not in any of the right contexts of other items either, so the input must be incorrect. The item will be chosen incorrectly, but the error will be detected later, so we only lose some early error detection.

Answer to Problem 9.30: There would be no LR state left to start the subsequent GOTO from.

Answer to Problem 10.1: The grammar is LL(1).

Answer to Problem 10.3: The number of openers depends on the length of the left spine, which is unknown even after the first node has been identified.

Answer to Problem 10.8: No. We put in the print statements so they would. Had we put them at the first possible place, just after the decision was taken, non-canonical orders would have resulted.

Answer to Problem 10.10: $\text{FOLLOW}_{\text{LM}}(A)$ is not a subset of FOLLOW(A) since it contains non-terminals. $\text{FOLLOW}_{\text{LM}}(A)$ is not a superset of FOLLOW(A) since it contains the tokens in FIRST(β) for any rule $B \rightarrow \alpha A \beta$.

Answer to Problem 10.12: `S→aAC|aBD|bAE; A→x; B→x; C→yC|c; D→yD|d; E→yE|e`, with input `axyyyye`. We end up with the reduced sentential form `a[A|B]E`, where neither `aAE` nor `aBE` is possible. So "Syntactic entity `E` cannot appear here".

Answer to Problem 10.17: If the right-hand side of the reduce rule is a substring S of the right-hand side of the shift rule, turn S into a rule $R \rightarrow S$. Now it is a reduce/reduce conflict between rules of equal length. It does imply more patching up of the parse tree.

Answer to Problem 11.3: The moment you discover the node is nullable (becomes recognized in the present prediction phase) append the parent node and back pointer nodes to the top list, and mark the node that this has happened. Upon connecting a new back pointer to a node marked this way, append the node pointed to to the top list. This discovers nullable left-recursive non-terminals dynamically.

Answer to Problem 11.5: Hint: The problem is insuring termination of step 1 in Section 11.1.1 in the face of left recursion and grammar loops. Most bottom-up tables protect against

left recursion (but not against hidden left recursion) and most do not protect against grammar loops.

Answer to Problem 12.1: *a.* The reversed grammar usually has much worse properties than the original. *b.* It does not solve sub*string* parsing.

Answer to Problem 12.2: The grammar will contain an undefined non-terminal A' and will need to be cleaned up.

Answer to Problem 12.4: It would not help much; you end up doing general CF parsing using the suffix grammar.

Answer to Problem 12.8: Nederhof and Satta [174] explain how to obtain an RDPDA for a given LR(0) grammar, but in doing the same for LR(1) one soon gets bogged down in details. A principled approach is needed.

Answer to Problem 13.1: See van Noord [221].

Answer to Problem 13.2: For each terminal t create rules $t \rightarrow t\ \phi$, where ϕ is some token not already occurring in the system. For each transition $n \xrightarrow{\varepsilon} m$ create a token ϕ_n_m. Now run the intersection algorithm as usual. Remove all occurrences of ϕ. This leaves a number of rules of the form $t\_p\_m \rightarrow t\_p\_n$; the occurrence of a t_p_m in a right-hand side means the occurrence of a transition $p \xrightarrow{t} n$ followed by a transition $n \xrightarrow{\varepsilon} m$. *Project*: extend this algorithm to handle ε-transitions from the start states.

Answer to Problem 13.8: The complexity is $O(n^3 \log n)$.

Answer to Problem 14.1: Discard, as this result cannot be combined with the result of the left neighbor.

Answer to Problem 14.7: Since (A,i,j) is realizable, there must be realizable CYK hypotheses (C,i,l) and (D,l,j) with $A \rightarrow CD$. Clearly, $size(C,i,l) \leq 2^k$ and $size(D,l,j) \leq 2^k$. Now, if $size(C,i,l) \leq 2^{k-1}$ and $size(D,l,j) \leq 2^{k-1}$) then (A,i,j) is its own critical hypothesis. Else, if $size(C,i,l) > 2^{k-1}$, $size(D,l,j) < 2^{k-1}$ (or else $size(A,i,j)$ would be $> 2^k$), and vice versa. Now suppose $size(C,i,l) > 2^{k-1}$. The reasoning is identical for $size(D,l,j) > 2^{k-1}$. We can now recurse, and use the same reasoning to find the critical hypothesis for (C,i,l), which will then also be the critical hypothesis for (A,i,j). This process will stop, because $size(C,i,l) < size(A,i,j)$.

Answer to Problem 14.8: Hint: add nodes $A \rightarrow BC'_{i,j,k}$ for all grammar rules $A \rightarrow BC$ and all $0 \leq i < j < k \leq n$. Are these AND-nodes or OR-nodes? Where do they get their input from?

Answer to Problem 15.1: Start with $S \rightarrow At_1t_2 \cdots t_kY$. Let A produce $t_1^{n-1}X^{n-1}$ where the Xs act as "sparks". The sparks can move to the right, produce a t_i on each (t_i, t_{i+1}) boundary, and are extinguished by the Y, producing the final t_k.

Answer to Problem 15.3: See [244].

Answer to Problem 15.7: See, for example, J. Higginbotham, *English is not a Context-Free Language*, Linguistic Inquiry, **15**, 2, (1984), pp. 225-234. Or: J. Bresnan et al., *Cross-Serial Dependencies in Dutch*, in W. J. Savitch et al., "The Formal Complexity of Natural Language", pp. 286-319, (1987).

Answer to Problem 15.11: Without the restriction it is trivial to give a coupled grammar for the language a^{n^2}: `S->Q1`; `Q1,Q2->Q1Q2Q2a,Q2a|a,a`. Can this also be done *with* the restriction?

Answer to Problem 15.12: For example: 1. Grammars embody declarative specification, in the functional paradigm; recognizers embody algorithmic specification, in the imperative paradigm. It is generally acknowledged that the functional paradigm is "higher", more

"evolved" than the imperative one. 2. Nobody knows what exactly the recognition power of recognition systems is; 3. There is 50 years of knowledge on grammars.

Answer to Problem 15.13: *a*. `abbcc`, or simply `bc`. *b*. `S←A!b&a*C`.

Answer to Problem 15.14: Basically, develop `P` by substitution, and discard an alternative `A` when FIRST(`A`) does not contain `a`. For the ensuing complications with left recursion and empty productions see Greibach [7].

Answer to Problem 15.15: *b*. 1. Determine which subexpressions recognize the empty string, for example by propagating this property bottom-up. 2. Construct a directed graph G in which the subexpressions are the nodes and arrows point from each node to the nodes it depends on directly. For a subexpression $\alpha A\beta$ where α recognizes ε, this is A. 3. Do a topological sort on G.

Answer to Problem 15.17: `S←a!./aaaa!./aaaBaS B←S&ε/aBa`

Answer to Problem 16.1: 1. Yes, but a full LL(1) parser would be better, since it remembers more left context.

Answer to Problem 16.4: Reducing as much as possible usually enlarges the set of acceptable tokens, thus making it more likely to find a correction. Predicting as much possible only reduces that set.

Answer to Problem 16.5: BRC and BC grammars allow parsing to be resumed at any subtree that has only terminals as leaves; there are many of these in a text of any length. With BCP grammars there may be only one such subtree to continue the parsing; if precisely that subtree is damaged by the error, there is no way to resume parsing.

Answer to Problem 16.6: Actually we do not need the prediction; we only need the set of symbols in it. We implement this set as an array of counters indexed by symbols. Upon entry to an alternative in a routine R_i we increment the counters for the symbols that that alternative would stack, and upon leaving we decrement the counters. If a counter for a symbol X_j is not zero, X_j would be in the prediction if it existed, and its FIRST set can be added to the acceptable set.

Answer to Problem 16.7: We compute the first insertion y_1, either as `cheapest_insertion`(X_1, a) if it exists or as `cheapest_derivation`(X_1); this may or may not make a acceptable. We insert y_1 and restart the parser. If the insertion y_1 did not make a acceptable, the parser again gets stuck, on X_2 this time. We repeat the process.

Answer to Problem 17.1: It blocks any repeated parsing of the same string with the same grammar rule, so it does not see or report it.

Answer to Problem 17.5: The function `q` would be called even if `p` failed, which would cause the parser to loop on any recursion, rather than on left recursion alone!

Author Index

Page numbers in roman indicate pages referring to the author; page numbers in *italic* refer to annotations of works by the author. Names and initials are indexed as shown in the publications in question.

Aasa, A, 272, *596*
Abney, S. P, 352, *638*
Adams, E. S, 263, *593*
Aguiar, M. Á. P, *604*
Aho, A. V, 162, 303, 316, 529, *598, 599*, *623*, *630, 631*, 639, 647
Akker, R. op den, 377, 378, *608*, *613*
Al-Hussaini, A. M. M, 285, *587*
Alblas, H, 470, 613, *613*
Albro, D. M, 439, *611*
Alonso Pardo, M. A, 314, 390, *603*, *636*
Anderson, S. O, 539, 540, *625*, 626, *626*
Anderson, T, 302, 303, 313, *586*
Augusteijn, L, *632*
Aycock, J, 217, 218, 318, 390, 551, 583, *583*, 603, *603, 604*

Backhouse, R. C, 539, 540, *600*, *625, 626*
Baker, T. P, *586*, 589
Bal, H. E, *643*
Bańko, M, 601, *601*
Bar-Hillel, Y, 425, *611*, *638*
Bar-On, I, 447, 471, *612*
Barnard, D. T, 506, 585, *585*, *619*
Bates, J, 409, 414, 415, 418, 447, 549, 573, *610*, 611, 629
Bauer, F. L, 584, *594*
Bell, J. R, 272, *595*
Bermudez, M. E, 311, 312, 315, 323, 333, 336, 338, 340, *588–590*
Bertsch, E, 272, 299, 319, 401, 407–409, 414, 418–421, 546, 591, *591*, *595*, *610*, 628, *628*
Bhamidipaty, A, 552, *633*
Bhate, S, 14, *642*
Billington, D, 44, *640*
Billot, S, 91, 426, 582, *601*
Birman, A, 506, 510, 519, *615*, 622
Blank, G. D, 141, 637, *637*
Bod, R, 100, *632*
Bolognesi, R, 391, *602*
Bouckaert, M, 223, *579*, 601
Boyer, R. S, 162, *598*
Bratley, P, 141, *636*
Brent, R. P, 460, 462, 463, *612*
Brown, P. J, 523, *626*
Brüggemann-Klein, A, 141, *599*
Bryant, B. R, 484, *618*
Brzozowski, J. A, 148, *597*, 634
Bugge, E. H, *626*
Burke, M. G, 532, 590, 627, *627*

Cabrero Souto, D, *603*
Carter, L. R, 552, *631*
Cerecke, C, 540, *628*
Chang, C.-H, 540, 587, *587*, 588, 626, *626*
Chang, J. H, 458, *612*
Chapman, N. P, 317, *588*
Charles, P, 314, 532, *590*, *627*, 648
Cherry, L. L, 316, *634*
Chester, D, 352, *637*
Choe, K.-M, 318, 540, 587, *587*, 588, *588*, 590, *590*, 626, *626*, *628*
Chomsky, N, 14, 38, *638*
Clark, C, 550, *633*
Cleaveland, J. C, 479, *617*
Cocke, J, 77
Cohen, J, 192, *581*, *631*
Cohen, R, 328, 333, *586*, 587, 606, 640
Colmerauer, A, 358, 359, 595, *595*, *606*
Conway, M. E, 623, *630*
Conway, R. W, 543, *623*
Cook, S. A, *639*
Corasick, M. J, 162, *598*
Corchuelo, R, 540, *628*
Cordy, J. R, 506, 585, *585*, *619*
Cormack, G. V, 276, 364, 409, *607*, *610*, *627*
Cormen, T. H, 76, *643*
Čulik, II, K, 328, 333, *586*, 587, 606, 640

De Vere, L, *628*

Degano, P, 318, *629*
Dembiński, P, 482, *617*
Demers, A. J, 343, *592*, 603
Dencker, P, 146, 285, *630*
DeRemer, F. L, 302, 306, 307, 311, 314, 532, 586, *586*, 587, *587*, 588, 590, *624*
Deudekom, A. van, 391, 398, 409, 541, *602*, *627*
Deussen, P, 68, 84, *580*
Dewar, H. P, 141, *636*
Dion, B. A, 390, *629*
Dömölki, B, 578, *578*
Dürre, K, 146, 285, *630*
Dwyer, B, 523, *625*

Earley, J, 4, 206, 290, *578*, 580, 601
Economopoulos, G. R, *604, 605*
Edupuganty, B, 483, *618*
Eickel, J, 276, 594, *594*
Erk, K, 567, *636*
Evans, W. S, 568–570, *635*
Eve, J, *586*

Farré, J, 327, 333, 338, 358, 372, *591*, *604*, *609*
Fischer, C. N, 444, 537, 538, *612*, 625, *625*, 626, *626*, 630
Fisher, A. J, *618*, *620*
Fisher, G. A, 482, 532, 590, 627, *627*
Floyd, R. W, 276, 278, *594*, 595, *605*, 606
Ford, B, 506, 507, 509, 510, 622, *622*
Fortes Gálvez, J, 318, 325, 327, 333, 358, 372, 390, *590, 591*, *604*, *609*, 648
Foster, J. M, 252, *641*
Fraser, C. W, *635*
Friede, D, 354, 356, 357, 606, *606*
Friš, I, 503, *615*, 616
Frost, R. A, 564, 566, *631, 632*
Fu, K.-S, 530, *576*, *625*
Fuji, M, *620*

Gazdar, G, 637, *637*
Geffert, V, 19, *640*
Gerevich, L, 482, *618*
Gilbert, P, 506, *614*
Glanville, R. S, 571, 573, 574, *630*
Goeman, H, 418, 610, *611*
Goldschlager, L. M, 460, 462, 463, *612*
Gonser, P, *596*, *625*, 626
Goodrich, M. T, 76, 78, *643*
Graham, S. L, 212, 216, 224, 525, 530, 547, 571, 573, 574, *579, 580*, 619, 624, *624*, *630*
Grau, A. A, 563, 574, 584, *630*
Gray, J. N, 272, *595*
Greibach, S. A, 61, 100, 101, 168, 173, 391, *577*, 639, *639*, 650
Griffiths, T. V, 84, *577*
Grimm, R, 513, *622*
Grune, D, 253, 259, 481, 484, 488, 618, *618*, *620*, *634*, *643*

Haines, L. H, 422, 423, *639*
Hammer, M, 252, *642*
Harrison, M. A, 212, 216, 224, 272, 547, *579, 580*, *595*
Hartmann, A. C, *624*, 625
Hays, D. G, 77, *576*
Heckmann, R, 259, *585*
Heering, J, 318, *589*
Heilbrunner, S, 317, 338, 514, 516, *587*, *619*, *640*
Henderson, D. S, *595*, 596
Hendrickson, K. J, 448, *613*
Heuft, J, 146, 285, *630*
Hext, J. B, 204, 578, *578*
Hickey, T. J, 192, *581*, *631*
Hopcroft, J. E, 21, 72, 119, 158, 168, 338, *639*, 640
Horning, J. J, 523, *586*, *595*, *624*
Horspool, R. N, 217, 218, 224, 318, 390, 551, 552, 583, *583*, *589*, *603, 604*
Hotz, G, 502, *614*, *621*
Huens, J, *585*, *624*
Hunt, H. B, 252, *642*
Hussain, S. S, *604*
Hutton, G, 564, 566, *631*, 632
Hutton, M. D, 372, *608*
Huybrechts, M, *585*, *624*

Ibarra, O. H, 458, *612*
Ichbiah, J, 273, *595*
Irons, E. T, 352, *576*, 630
Ives, F, *587*, 588, *588*

Jackson, Q. T, 160, 506, 516, 517, *622*, 623, *623*
Jacobs, C. J. H, 253, 259, *634*, *643*
Janoušek, J, *604*
Janssen, W, 470, *613*
Jarzabek, S, 257, *585*
Jeffery, C. L, 523, 544, *629*
Johnson, M, 352, 566, *632*, *638*
Johnson, S. C, 303, 316, *630*, *634*
Johnstone, A, 185, 188, 318, 390, *583*, *592*, 604, *604*, 605, *605*, 634, *634*
Jones, D. W, 553, *598*
Joshi, A. K, 492, 497, 500, 518, *616*, 619, *619, 620*

Kak, S, 14, *642*
Kannapinn, S, 302, 318, 327, *591*
Kantorowitz, E, 523, *626*
Karp, R. M, 162, *598*
Kasami, T, 77, *578*, *620*
Kastens, U, 488, *617*
Kay, M, 232, 377, *579, 580*, 581, *607*
Kernighan, B. W, 316, *634*
Kilbury, J, 232, *580*
Kim, I.-S, 540, *628*
Kim, J, 452, 453, *614*
Kipps, J. R, 390, *602*
Kirchhoff, L. W, 253, *630*
Kleene, S. C, *597*
Klein, E, *637*
Klint, P, 318, *589*
Knuth, D. E, 290, 300, 358, 365, 485, 563, *584, 585*, 586, *615*, 618
Kooiman, P, 391, 398, 409, 541, *602*, *627*
Koorn, W, 408, 547, *610*
Korenjak, A. J, 300, 586, *586*, 629
Koskimies, K, 553, 564, *631*
Koster, C. H. A, 485, 488, 490, *614, 615*, *619*
Král, J, 252, *593*
Krawczyk, T, 257, *585*
Kristensen, B. B, 587, *587*
Kruijff, G. M, 567, *636*
Krulee, G. K, 477, *620*
Kruseman Aretz, F. E. J, 212, 319, *582*, *588*, 589, *632*
Krzemień, R, 139, *598*
Kulkarni, S. R, 504, 506, *621*
Kuno, S, *576*, 633

Kurapova, E. W, 570, *632*
Kurki-Suonio, R, 255, *584*

LaLonde, W. R, 318, 586, *586*, 587, *587*
Lancaster, R. L, *642*
Lang, B, 91, 382, 401, 426, 582, *582*, 601, *601*, *609*, *611*
Langendoen, K. G, *643*
Lankhorst, M, 390, 448, 452, 471, 547, *602*, *613*
Laor, H, 523, *626*
Laurikari, V, 147, *599*
Lavie, A, 409, 414, 415, 418, 422, 447, 549, 573, *603*, *610*, 611, 629
Lee, L, 99, *583*
Leermakers, R, 564, 600, *600*, *632*, 633
Leiserson, C. E, *643*
Leo, J. M. I. M, 226, 548, *582*
Lepistö, T, 503, *616*
Lesk, M. E, 32, *634*
Levy, L. S, *616*
Levy, M. R, 271, *595*, 596
Lewi, J, 259, *585*, *624*
Lewis, II, P. M, 352, 379, *592*
Lindstrom, G, 414, *629*
Liu, L. Y, 53, *639*
Ljunglöf, P, 566, *633*
Loeckx, J, 275, 276, *595*
Logothetis, G, 312, 340, *589*
Lomet, D. B, 346, *592*
Lucas, P, 563, *584*
Łukasiewicz, A, 139, *598*
Lyon, G, 529, 611, *623*

Madhavan, M, 376, *608*
Madsen, O. L, 485, 490, 587, *587*, *618*
Małuszyński, J, 481, 482, *617, 618*
Mannucci, S, 318, *629*
Matsumura, T, *620*
Mauney, J, 625, *626*
McClure, R. M, *614*, 615
McKeeman, W. M, 275, *595*
McKenzie, B. J, 316, 540, *590*, 628, *628*
McLean, P, 224, *583*
McNaughton, R, 160, *597*
Mealy, G. H, 160, *596*
Meersman, R, 479, *617*
Meertens, L. G. L. T, 485, *614*
Melichar, B, *604*
Merrill, G. H, 381, *603*
Metsker, S. J, 564, *633*
Michie, D, 133, *642*
Mickunas, M. D, 299, 532, *624*, *642*
Milton, D. R, 253, 537, 538, *625*, 626, 630, *630*
Modry, J. A, 532, *624*
Mojana, B, 318, *629*
Moore, E. F, 160, 597, *597*
Moore, J. S, 162, *598*
Morawietz, F, 567, *636*
Morimoto, S.-I, 318, *591*
Morse, S, 273, *595*

Nakata, I, 318, *588*, 590
Nederhof, M.-J, 134, 192, 288, 299, 319, 333, 353, 377, 390, 397, 401, 408, 414, 418, 420, 421, 490, 492, 500, 546, *582, 583*, *591*, 592, *593*, *600*, *603*, *608*, *610*, 619, 621, *621*, 628, *628*, *632*, *641*, 647, 649
Nijholt, A, 84, 257, *585*, *600*, *613*
Noord, G. van, 91, 378, *608*, *611*, 649
Norvig, P, 133, *631*, 632
Nozohoor-Farshi, R, 389, 591, *602*, 604
Nuutila, E, 97, *643*

Oettinger, A. G, *576*, 633
Ohsawa, I, 452, *612*
Okhotin, A, 514, 516, *621–623*
Ostrand, T. J, 147, *598*
Oude Luttighuis, P, *613*

Palis, M. A, 458, *612*
Park, J. C. H, 587, *587*, 588, *588*, 626
Parr, T. J, 257, *585*
Paul, M, 594, *594*
Paull, M. C, 147, *598*
Pemberton, S, 535, *625*
Pennello, T. J, 302, 306, 307, 311, 314, 532, 552, 587, *587*, 588, *588*, 589, 590, *624*
Peralta, J. C, 573, *636*
Pereira, F. C. N, 333, 582, *635*, 641, *641*
Pérez, J. A, *628*
Perles, M, 425, *611*, *638*
Peterson, T. G, 529, *623*
Petrick, S. R, 84, *577*
Peyton Jones, S. L, 272, *596*
Piastra, M, 390, *602*
Pirotte, A, 223, *579*, 601
Pitsch, G, 502, *621*
Poel, M, *613*
Pollard, C. J, 231, 377, *607*
Poplawski, D. A, 257, *585*
Proebsting, T. A, 552, *633*
Prolo, C. A, 500, *621*
Proudian, D, 231, 377, *607*
Pullum, G, *637*

Quiring, S. B, 537, *625*, 626, 630
Quong, R. W, 257, *585*

Ra, D, 452, 453, *614*
Rabin, M. O, 162, *598*
Rai, S, *608*
Ramakrishna, U, *608*
Raoult, J.-C, 582, *582*
Rayward-Smith, V. J, 12, *640*
Rekers, J, 318, 408, 547, *589*, *602*, 603, *610*, *632*
Resnik, P, 352, *638*
Révész, G. E, 18, 21, 53, 72, 640, *640*
Rhodes, S. P, 525, 530, 624, *624*
Richter, H, 276, 391, 540, 542, 602, 610, *626*, 627
Rivest, R. L, *643*
Roberts, G. H, 319, 588, *588*, 589, *589*, 590, *590*
Roberts, P. S, 204, 578, *578*
Roche, E, 163, *599*
Röhrich, J, 535, 537, *625*
Rosenblueth, D. A, 573, 635, *635*, 636, *636*
Rosenkrantz, D. J, 252, 352, 379, *592*, *642*
Rowland, B. R, 253, *630*
Rozenberg, G, 479, *617*
Ruckert, M, 276, 533, *611*, *628*
Ruiz, A, *628*
Rus, T, 134, 135, *581*
Rußmann, A, 476, *621*
Ruzzo, W. L, 212, 216, 224, 547, *580*

Ryabko, B. Y, 570, *632*
Rytter, W, 100, 460, 468, *582*, *612*

Sag, I, *637*
Sakai, I, 77, 112, *576*
Salomon, D. J, 364, *607*
Samelson, K, 584, *594*
Sarbo, J. J, 288, 390, 490, 492, *591*, *632*
Sassa, M, 318, *588*, 590, *591*
Satta, G, 134, 377, 420, 421, 500, *583*, *603*, *607, 608*, *621*, 649
Schabes, Y, 500, 600, *600*, *619, 620*, 621
Schimpf, K. M, 311, 315, 333, 336, 338, *588, 589*
Schlesinger, S. I, 263, *593*
Schmidt, E, 32, *634*
Schmitz, L, 514, 516, *619*
Schmitz, S, 338, *640*
Schneider, V. B, *642*
Schuler, P. F, 514, *616*
Scott, E, 185, 188, 318, 390, *583*, 592, *592*, 604, *604*, 605, *605*, 634, *634*
Sedgewick, R, 78, *643*
Seki, H, 502, 611, *620*
Sethi, R, 303, *631*
Seyfarth, B. R, 323, *590*
Shamir, E, 425, *611*, *638*
Shankar, P, 506, *608*, *621*
Share, M, 316, *634*
Sheil, B, 77, 110, 128, 133, *579*, 601
Sheridan, P. B, 339, *594*
Shieber, S. M, 580, *637*
Shin, H.-C, 318, 590, *590*
Sikkel, K, vii, 232, 377, 378, 448, 452, 454, 468, 470, 471, *600*, *608*, *613*
Sintzoff, M, 479, *615*
Snelling, M, 223, *579*, 601
Soisalon-Soininen, E, 379, *593*
Sperber, M, 566, 600, *633*
Sperberg-McQueen, C. M, 141, 598, *634*
Srikant, Y. N, 447, *612*
Stein, C, *643*
Stirling, C. P, 535, 626, *626*
Stock, O, 377, *607*
Stone, R. G, 285, *587*
Sudkamp, T. A, 12, 80, *640*
Szydlowski, B, 564, *632*
Szymanski, T. G, 365, 367, 372, *606*

Tai, K.-C, 360, 362, 363, 538, *607*, *625*, 626
Takahashi, M, *616*
Tamassia, R, 76, 78, *643*
Tanaka, E, 529, *576*, *625*
Tarjan, R. E, 97, 311, *630*
Thiemann, P, 566, 600, *633*
Thompson, K, 149, *598*
Thompson, S, 566, *643*
Thorne, J. P, 141, *636*
Tomita, M, 84, 210, 382, 383, 385, 387, 422, 578, 591, 601, *601*, 602, *603*, 646
Torii, K, *578*
Toro, M, *628*

Ukkonen, E, 299, 379, *587*, *593*
Ullman, J. D, 21, 72, 119, 158, 168, 303, 316, 338, 506, 510, 519, *599*, *615*, 622, *630, 631*, *639*, 640, 647

Unger, S. H, 77, 104, 107, *578*
Uzgalis, R. C, 479, *617*

Valiant, L. G, 98, *579*, 583
Vijay-Shankar, K, 497, 518, 619, *619*, 621
Vilain, M, 573, *635*
Vilares Ferro, M, 314, 390, *603*, *629*, *636*
Vishkin, U, 447, 471, *612*
Vlaminck, K. de, *585*, *624*
Voisin, F, 582, *582*

Waite, W. M, 552, *631*
Warren, D. H. D, 582, *635*
Warshall, S, 97, *642*
Watson, B. W, 162, *599*
Watt, D. A, 485, 490, *618*
Wattel, E, 602
Weber, H, 273, 274, 595, *595*
Wegner, L. M, *617*, 618, 620
Weiner, P, 53, *639*
Weyuker, E. J, 147, *598*
Whitney, M, 552, *589*
Wijngaarden, A. van, 477, 479–481, *615*, 616, *616, 617*
Wilcox, T. R, 543, *623*
Williams, J. H, 277, 606, *606*
Williams, M. H, 271, 596, *596*
Wirén, M, 232, *581*
Wirth, N, 273, 274, 595, *595*
Wolpe, H, 263, *594*
Wood, D, 141, *599*
Woods, W. A, 56, *637*
Wortman, D. B, *595*
Wright, R. N, 333, 641, *641*

Yamada, H, 160, *597*
Yeatman, C, *628*
Yli-Jyrä, A, 333, *641*
Yngve, V. H, 492, *636*
Yonezawa, A, 452, *612*
Younger, D. H, 77, 112, *577*

Žemlička, M, 252, *593*
Zwiers, J, *613*

Subject Index

Page numbers in **bold** refer to pages where a definition of the indicated term can be found.

$\xrightarrow[l]{}$, **38**
$\xrightarrow[l]{*}$, **38**
$\xrightarrow[l^R]{}$, **38**
$\xrightarrow[l^R]{*}$, **38**
$\xrightarrow{*}$, **38**
$\aleph_0$, 12
$\aleph_1$, 12
$\cap$, **52**
$\cup$, **52**
$\neg$, **52**
$\nrightarrow$, 607
$\not\vdash$, 607
§-calculus, 160, 506, **516**, 622, 623
$\leqslant$, **273**
$\doteq$, **267**
$\geqslant$, **267**
$\lessdot$, **267**
ε-LR, 591
ε-LR(0), **287**
ε-closure, **148**
ε-free, **27**, 40
ε-move, **147**, 293, 525
ε-rule, **27**, 104, 147, 182, 221, 242, 295
ε-transition, **147**, 163, 281
2-form, 497
2-form grammar, **579**
2DPDA, 423, 639
4-tuple, **14**

A-BNF, 623
acceptable set, 627
acceptable-set, **533**
acceptable-set error recovery, **533**
accepting state, 142, 168
accessible non-terminal, **51**
ACTION table, **284**
ACTION/GOTO table, **285**
active edge, **227**
ad hoc error recovery, **542**
adaptable grammars, *see* dynamic grammars
adjacency restriction, 607
adjoining, **494**
affix, **488**, 619
affix grammar, 485, **488**, 615
agenda, **228**, 581
agent parsers, **447**
AGFL, **490**, 632
Aho and Corasick bibliographic search algorithm, **162**, 598
algemeines LR, **592**
ALGOL 60, 27, 514, 576, 594
ALGOL 68, 476, 615
alphabet, **6**, 578
alternative, **15**
ambiguity, **63**, 144, 316, 599, 630, 634
ambiguity rate, 578
ambiguity test, 63, 338
American Indian languages, 492
amusing, 619
analysis stack, **170**
analytic grammar, 506, 614
ancestors table, 602
AND-node, **88**
AND-OR tree, **88**
AND/OR form, **135**
angle brackets, 27, 619
ATN, **56**, 604, 620, 637
attribute, **54**, 254, 485, 549, 630, 631
attribute evaluation rule, **485**
attribute grammar, 485, **485**, 588, 615, 643
Augmented Transition Network, *see* ATN
auxiliary tree, **493**

backreference, **159**
backslash, 508
backtracking, **78**, 176, 182, 201, 579, 600, 614
backtracking recursive descent, 576
Backus-Naur Form, *see* BNF
backward move, **530**, 624
backward/forward move, **530**
BC(*h*)DR(0), 609

BC(*m*,*n*), **276**
BC(1,1), 409
BCP, **277**, 606, 628
BCP(*m*,*n*), 611
biconnected, **630**
binary form, **214**, 603
binary right-nullable, 604
binary tree, **62**, 214, 577, 612
bison, 301, 633
bitvector, 578
blind alley, **17**, 35, 480
BMM, *see* Boolean matrix multiplication
BNF, **27**
Boolean circuit, **453**
Boolean closure grammar, **514**
Boolean grammar, **514**, 619, 622, 623
Boolean matrix multiplication, 97, **98**, 500, 583
bottom-up, 581
bottom-up parsing, **66**
bounded-context, **276**, 532, 595, 606, 610, 628
bounded-context parsable, **277**, 533, 606, 628
bounded-left-context, 595
bounded-right-context, 85, **276**, 525, 532, 593, 595, 606, 611, 642
BRC(*m*,*n*), **276**, 611
breadth-first production, **35**, 618
breadth-first search, **77**, 170, 204, 235, 278, 382, 547, 601, 643
BRNGLR, 605
Brzozowski derivative, 597
built-in conflict resolver, 315
Burushaski, 490
byacc, 603

C(*k*), **353**
cancellation parsing, 85, **192**, 353, 483, 593
cancellation set, **192**, 353
Cantor, 12
cascaded grammar, **637**
category (GPSG), **637**
cellar, **594**
chain rule, **113**, 316
channel algorithm, **303**
character-pair error, **525**
chart, **226**, 581
chart parser, 635
chart parsing, 85, **226**
Chomsky hierarchy, **19**, 40, 617, 638
Chomsky Normal Form, **116**, 454, 460, 577, 638
closed part, 137, 138
closure, 587
closure algorithm, **50**, 95, 114
CLR(1), **592**
CNF, *see* Chomsky Normal Form
Cocke-Younger-Kasami, *see* CYK parser
combinator, **564**, 631, 632
combinator parsing, **564**
Common-Prefix, 582
Compact LR, **592**
compiled recursive descent, **185**, 253, 563, 584
complement, 52, 152
complete automaton, **152**, 290
completed item, *see* inactive item
compound value (Prolog), **189**
concatenation, 24
condensation phase, **530**, 624
configuration, 633
conflict resolver, **253**, 254, 315, 421, 485, 590, 627, 634
conjunctive grammar, 621
connectionist network, 453, **453**
connectionist parser, 453
consistent substitution, **479**, 618
constraint programming paradigm, 567
context-free grammar, **23**
context-sensitive, **20**
context-sensitive grammar, 20, 23, 73, 545, 638
continuation
  error recovery, **535**
  functional programming, **183**
continuation grammar, **536**
continuous grammar, **533**, 628
control grammar, **502**, 621
control mechanism, 69, 81, 265
control sequence, **502**
control word, **615**
Cook's Theorem, 639
core of an LR state, **300**
coroutine, 86, 630
correct-prefix property, **248**, 521, 524, 540
correction phase, **530**
counter language, **638**
coupled grammar, **500**
critical hypothesis, **467**
critical step, **466**
cross-dependencies, **492**
cross-reference problem, **482**, 617–620
cubic space dependency, **87**
cubic time dependency, **71**, 80, 526, 547, 577
cycle, **16**
CYK parser, 70, 85, **112**, 212, 576, 577, 600

dag, **16**, 34, 88, 387, 475, 490
data-oriented parsing, 100, 632
DCG, 85, **189**, 193, 553, 573, 618
De Morgan's Law, 53, 153
decomposition of an FSA, 154
deficient LALR algorithms, 302
definite clause, 188, **189**, 482, 635
Definite Clause Grammar, *see* DCG
definite event, **597**
depth-first search, **77**, 105, 170, 235, 553, 630, 643
derivation tree, 38
derivative, **597**
derived attribute, 55, **486**
derived information, **54**
description language, **606**
deterministic automaton, **82**, 283, 293, 524
deterministic cancellation parsing, 85, 353
deterministic finite-state automaton, *see* DFA
deterministic grammar, 333
deterministic language, **299**
deterministic parser, **235**, 238, 247, 299, 525
DFA, **145**
DFA minimization algorithm, **157**
difference list, 636
directed graph, **16**
directed production analyser, 391, **577**
directional parser, 93
directional parsing method, **76**
directly-reads, **309**
disambiguating rule, **252**, 630, 634
discontinuous constituents, 636
Discriminating Reverse, *see* DR
distfix operator, **272**, 596

document conversion, 316, 634
don't-care entry, 630
dot, 180, **206**, 280, 624
dot look-ahead, **298**, 366
dot right context, 322, 323
dotted look-ahead, 366
double-dotted item, 608
DR, 85, **325**, 390, 604, 609
DR automaton, **326**
DR parsing, 327, 590
DR(*k*), 358, 590
Dutch, 140, 492
dynamic grammars, **476**, 621
dynamic programming, 643

EAG, 485, 490, 620
eager completion, 225
Earley deduction, 635
Earley item, **206**, 207, 280, 529, 623
Earley parser, 70, 85, **206**, 290, 407, 452, 529, 546, 547, 600, 625
  bottom-up, 452
Earley set, 212
Earley suffix parser, **408**
Earley-on-LL(1) parser, 409
early error detection, 299
EBNF, **28**, 318, 591
edge, **16**
elementary tree, **493**
eliminating ε-rules, 119, 175
eliminating left recursion, 174, 593, 641
eliminating unit rules, 119, 125, 175, 316
ELR, 582
empty language, 41
empty string, 41
empty word, **9**
end marker, 13, **172**, 236, 266, 535
end-of-input, **94**, 219, 324
envelop, **328**
erasable symbols, **53**
error correction, **522**, 630, 634
error detection, 475, **521**
error detection point, 526, **526**, 624
error handling, 475, 488
error interval, **542**
error production, 529, **542**, 623
error recovery, 500, 521, **522**, 540, 600
error repair, 475, **522**, 600, 627
error reporting, 475
error symbol, **526**, 627
error token, 542, **543**
essential ambiguity, **63**
EULER, 595
evaluation rule, *see* attribute evaluation rule
event, **597**
EXCLUDE set, 578
exclusion rule, 607
Exhaustive Constant Partial Ordering property, **637**
expanding an item, 289
exponential explosion, 204
exponential time dependency, **71**, 110, 116
expression-rewriting code generation, **571**
Extended BNF, *see* EBNF
extended consistent substitution, 618, **618**
extended context-free grammar, 28, **28**, 259, 578
extended LL(1), 259, 585, 624, 634
extended operator-precedence, **595**
extended precedence, **274**, 525

factorization of an FSA, 154
fast string search, 161, 598, 599
feature, 496, **637**
feedback arc set, 603
FILO list, *see* stack
finite lattice, 490
finite-choice, 370, 496
finite-choice grammar, **33**, 473, 479
finite-state automaton, 137, **142**, 278, 330, 332, 426, 524, 585, 586
finite-state grammar, **30**, 473
FIRST set, 220, 239, **239**, 242, 621
FIRST_k set, 239, 254, **254**, 255, 256
$\text{FIRST}_{\text{ALL}}$ set, **273**
FIRST_{OP} set, 270, **270**
FIRST/FIRST conflict, **241**, 633
FIRST/FOLLOW conflict, **246**, 633
fixed-mode logic, 635
FL(*k*), **598**
flattening (LR(0) automaton), 641
Floyd production, 277, **277**, 594
FMQ error correction, **537**, 538, 625
FOLLOW set, 244, **245**, 312, 353, 534, 621
FOLLOW_k set, 255, **255**, 256
$\text{FOLLOW}_{\text{LM}}$ set, 361, **361**
FOLLOW-determinism, 583
FOLLOW-set error recovery, 534, **534**
FOLLOW/FOLLOW conflict, **246**, 647
forest of tree-structured stacks, **414**
forward move, **530**
FPFAP(*k*), **606**
frame language, 573, 635
free of hidden empty notions, 617
French, 505
FSA, *see* finite-state automaton
full backtracking top-down parsing, 501
full-LC(1), **348**
full-LL(*k*), 256
full-LL(1), 248, 348, 525, 549
fully parenthesized expression, **266**, 594
functional programming, 481

gap, 200, 278, 366
general hyperrule, **619**
Generalized cancellation, 85, 397
Generalized LC, 85, 397
Generalized LL, 85, 391, 602, 627
Generalized LR, 85, 263
generalized non-canonical parsing, 398
Generalized Phrase Structure Grammar, **637**
Generalized Recursive Descent Parsing, **185**
generalized TS, *see* gTS
generative grammar, **7**, 489, 616
Georgian, 492
German, 140
global error handling, **526**
GLR parser, 70, **382**, 448, 546, 547, 601
GLR suffix parser, **414**, 447
GNF, *see* Greibach Normal Form
GNU C, 184
goal
  Prolog, **188**
  top-down, **73**
GOTO table, **284**, 329

GPSG, *see* Generalized Phrase Structure Grammar
Graham–Glanville code generation, 376
grammar-based compression, **568**
grammar-type variable, 618
grammatical inference, **1**
graph, **16**
graph-structured stack, **387**, 451, 601, 604
GRDP, 583
Greibach Normal Form, **168**, 391, 577, 639
GRMLR, 604
GSS, *see* graph-structured stack
gTS, **616**
gzip, 568–570, 635

handle, **74**, 264
handle segment, **74**, 146, 269, 525
handle-finding automaton, 279, 291, 298, 374
hardest context-free language, 100, **639**, 640
Haskell, 564, 566, 633, 643
head grammar, **377**
head spine, 608
head-corner, 608
head-corner parsing, 231, **377**
hidden indirect left recursion, **174**
hidden left recursion, **174**, 288, 593, 603
high-water mark, 159
homomorphism, **640**
human natural language parsing, 352
Hungarian, 5
hypernotion, **479**, 617
hyperrule, **478**
hypothesis path, **465**

ID/LP grammar, 637
identification mechanism, 24
Immediate Dominance/Linear Precedence grammar, **637**
immediate error detection property, **248**, 299, 524, 525, 625, 626
immediate left recursion, **174**
immediate semantics, 550
in-LALR-lookahead, **310**
inactive item, **227**
inadequate state, **287**, 328, 381
includes, **309**
incremental parser generation, 318, 589, 632
incremental parsing, 318, 629
indexed operator precedence, 596
indirect left recursion, **174**
inference, **188**
inference rule, 51, 95, **96**, 227
infinite ambiguity, **49**, 86, 121, 395, 398
infinite symbol set, 484
infix notation, **65**
infix order, 350
inherently ambiguous, **64**
inherited attribute, 55, **486**
inherited information, **54**
initialization, **50**
insert-correctable, **538**
instantaneous description, **170**
instantiation, **188**
interpreted recursive descent, **185**
interpreter, 144, 283
intersection, **52**, 152, 425, 530
interval analysis, 627
invalidate, **623**

item, **206**, 290
item grammar, 640
item look-ahead, **298**
item right context, 322

Japanese, 577
Java, 553

kernel item, **207**, 289, 365, 590
keyword, *see* reserved word
kind grammar, 252, 593, **593**
KL-ONE, 635
Kleene star, **28**, 149, 164, 597
known-parsing table, **560**

LA(k)LR(j), **302**
LA(m)LR(k), 586
LALR, 590
LALR(k), 85, **301**, 587
LALR(k,t), 372, **608**
LALR(1), 587, 634
LALR(1) automaton, **301**, 589
LALR-by-SLR, **313**
LAR automaton, **333**
LAR parsing, 335
LAR(m), 85
LAR(m) parsing, **336**, 589
LAST set, 578
LAST_{ALL} set, **273**
LAST_{OP} set, **270**
lattice, 100, 381, 611, 619
lazy table construction, **553**, 631
LC, 582
LC(k), 85, 592, 593, 600
LC(1), **347**
least-error correction, **526**, 529, 544, 623
left context, 320, 321
left priority, **272**
left recursion, **24**, 133, 173, **174**, 192, 352, 553, 577
left recursion elimination, 252
left spine, **193**, 345
left-associative, 90
left-bound, 617
left-context, 253, 530
left-corner, **95**
left-corner derivation, **65**
left-corner parser, **345**
  bottom-up, 452
left-corner parsing, 345, 576, 637
left-corner relation, **352**
left-factoring, 252, **252**, 593, 642
left-hand side, **13**
left-regular, 404
left-regular grammar, **30**, 75, 154, 321, 610
left-spine child, **351**
left-to-right precedence, 595
leftmost derivation, **37**, 65, 124, 137, 165, 238, 343, 568
leftmost reduction, 199
lex, 32, 41, 149, 634
lineage, **42**
linear Boolean grammar, **622**
linear grammar, **30**, 579
linear time dependency, **71**, 81, 475, 626
linear time requirements, 327, 369
linear-approximate LL(k), **257**, 585
linearized parse tree, **65**, 637

Lisp, 566
list, 34
    Prolog, **189**
LL(k), 85, 254, **254**, 592, 593, 600, 625
LL(1), 238, 354, 409, 482, 537, 549, 581, 592, 618
LL(1) conflict, **251**, 641
LL(2), 255
LL-regular, 85, 585
LLgen, 391, 602, 627, 634
LLP(k), 592
LLR, *see* LL-regular
local error recovery, **533**
locally least-cost error recovery, **539**, 600, 625
locally unambiguous, 617
logic variable, **188**, 190, 441, 483, 567
long reduction, **415**
longest prefix, 94
look-ahead, **82**, 322, 348
lookback, 641
lookback, **308**
loop, **48**, 104, 127, 139, 174, 383
    hidden, 49
    indirect, 48
LPDA, 582
LR, 582
LR item, **280**
LR parse table construction algorithm, 289
LR state, 322
LR(k), 85, 280, 299, 585, 600, 642
LR(k) language, 299, 319
LR(k,∞), 85, 365, **365**, 606
LR(k,t), **371**, 606
LR($k > 1$), 299
LR($k \geq 1$), 299
LR(m,k), **586**
LR(0), 279, 285, **287**, 299, 301, 315, 382, 451, 586, 589
LR(0) automaton, **283**, 293, 300, 314, 383
LR(1), 83, 291, **293**, 364, 382, 586
LR(1) automaton, 291, 294, 300
LR(1) parsing table, 294
LR(1,∞), 365
LR(2), 299
LR-context-free, 372
LR-regular, 85, 226, 327, **328**, 586
LR-regular look-ahead, 329
LSLR(1), 362, 607

Manhattan turtle, **18**
marked ordered grammar, **504**
match, **73**
maximally supported parse forest, **400**
maze, 78, 82, 143
MCFG, 620
Mealy semantics, **160**
memoization, **133**, 422, 553, 566, 602, 608, 611, 622, 623, 632, 642, 647
memory requirements, 154
meta-deterministic language, **591**
metagrammar, **478**
metanotion, **478**, 616, 617
metaparsing, **456**
metarule, **478**
middle-out, 581
minimally completed parse tree, **400**
minimized DFA, **158**
minimum distance error handling, 623
mixed-strategy precedence, **275**, 595
modifiable grammars, *see* dynamic grammars
modifying the grammar, 81
Modula-2, 549
monad, 622, 643
Moore semantics, **160**
morphology, **7**
multi-way pointer, 92, 441
multiple context-free grammar, 611, **620**

NDA, *see* non-deterministic automaton
NDR, 609
negation, **52**, 53, 152
nested stack automaton, 623
NFA, **142**, 161, 281, 291, 304
NLALR(k), 372, **608**
NLALR(k,t), 372, **608**
NLR(k), **371**
node, **16**
non-canonical, **83**, 263, 277, 611
non-canonical LR, 365
non-canonical SLR(1), **361**
non-Chomsky grammar, 473
non-correcting error recovery, **541**
non-deterministic automaton, 69, **69**, 142, 200, 280, 296
non-directional parser, 93
non-directional parsing method, **76**, 103, 523
non-productive grammar, 48
non-productive non-terminal, 48, 120, 122
non-productive rule, 48, 427
non-terminal, **13**
NQLALR(k), 588
NQLALR(1), **311**, 587
NQSLR(k), 588
NQSLR($k > 1$), 315
NSLR(k), 85
NSLR(1), **361**, 607
nullable, **24**, 89, 217, 219, 297, 451
nullable LL(1) grammar, **625**
nullable non-terminal
    in BC and BRC, 277

occam, 618
open part, 137, 139, 165, 200, 321
operator grammar, **270**
operator-precedence, 268, **270**, 344, 548, 630
operator-precedence parser, 374
OR-node, **88**
oracle, 593
ordered attribute grammar, 488, **617**
ordered grammar, 502, 503
origin position, **207**
original symbol, **42**, 43

Packrat, 622
packrat parsing, **510**
palm tree, 630
panic mode, 534, **534**
Pāṇini, 642
parse dag, **88**, 622
parse forest, **87**, 482, 583
parse graph, **89**
parse table, **236**, 254, 256, 448, 549, 578, 631
parse tree, **62**, 621
parse-forest grammar, 49, 91, 110, 427, 451, 458, 512
parse-tree grammar, 427, 512
parser generator, **70**, 254, 285, 303, 538, 547, 630, 634

Parsing Expression Grammar, *see* PEG
parsing pattern, **606**
parsing schema, 600
parsing scheme, **606**
parsing table, **236**
partial evaluation, 566
partition, **157**, 355
Partitioned LL(*k*), 85
Partitioned LL(1), **354**
Partitioned LR, 344, **373**
Partitioned LR(*k*), 85
partitioned s-grammar, **357**
Pascal, 360, 542, 600
passive item, *see* inactive item
PDA, *see* pushdown automaton
PEG, 506, 509, **622**
phrase, 357
phrase language, **606**
phrase level error handling, **530**
phrase structure grammar, **15**, 637
PL/C, 623
PLL(0), 357
PLL(1)
    Partitioned LL, 344, **354**
PLR(*k*), 592, 593
polynomial time dependency, **71**, 110, 497, 560, 579
post-order, **65**, 201, 343
postfix notation, **65**
pre-order, **65**, 167, 343
precedence, 593
precedence functions, **271**, 549, 595
precedence parser, 85, 269, 525
precedence relation, 267, **267**, 525, 606
precedence table, **267**
predicate, 189, **489**
    ALGOL 68, **480**
    Prolog, **188**
predict, **73**
predicted item, **206**
prediction item, **207**
prediction stack, **167**, 176, 534
prefix, **93**
prefix notation, **65**
prefix-free, **182**
prettyprinter, 550
primitive predicate, 615
primordial soup algorithm, 470, 600
process-configuration parser, **447**
production, 581
production chain, 31, 592, **592**
production chain automaton, **346**
production expression, **592**
production graph, **16**
production rule, **16**
production step, **16**, 67, 239, 535, 592
production tree, **23**, 39, 54, 61, 65, 204, 637
Prolog, 80, 188, 314, 340, 482, 553, 567, 573, 608, 631, 636
Prolog clause, 188, 581
propagated input, 305
propagated look-ahead, 304
proper grammar, **49**, 536
PS grammar, **15**
pumping lemma for context-free languages, **44**
pumping lemma for regular languages, **45**
pushdown automaton, **167**, 319, 331, 420, 551, 592, 593, 610
quadratic time dependency, **71**, 211, 407
quasi-regular grammar, 139, **598**

Rats!, 622
RCA, 604
RDPDA, **420**, 422
reachable non-terminal, 51, **51**, 122, 148
read-back tables, **586**
reads, **310**
real-time parser, **71**, 637
recognition system, 488, 506, **507**
recognition table, 112, **117**, 126, 523, 610
recursion, **24**
recursive ascent, **319**, 552, 588–590, 630, 633
Recursive Call Automaton, **604**
recursive descent, 80, 85, 179, **181**, 254, 319, 553, 563, 584
recursive set, **36**
recursive transition network, **46**, 56
recursive-ascent, 633
recursively enumerable, **35**
reduce, **74**
reduce item, **206**, 282, 332, 361, 369, 582, 587, 607
reduce/reduce conflict, 279, **286**, 578, 607, 630
reduction error, **525**
Reduction Incorporated, 605
Reduction Incorporated Automaton, **604**
reduction look-ahead, **222**, 579
reduction-incorporated, **317**
regional error handling, **530**
register-vector grammar, 637, **637**
regular envelope, 328, 641
regular expression, **28**, 33, 147, 149, 269, 598, 634
regular grammar, **30**, 148
regular language, 45, 53, 139, 600, 636
Regular Pattern Parsable, **606**
regular right part grammar, **28**, 318, 582, 586, 588
relations algorithm, **306**, 587
reserved word, 33, 76
reverse finite-state automaton, 330
reversed scan, **456**
reversed tree, 391
RI, *see* reduction-incorporated, 605
RIA, 604
right context, 320, 322, 329, 628
right priority, **272**
right recursion, **24**, 582
right-associative, 101
right-bound, 617
right-hand side, **13**
right-nullable, **604**
right-nulled LR(1), 605
right-regular grammar, **30**, 75, 139
rightmost derivation, **37**, 65, 124, 137, 199, 200, 343
rightmost production, 199, 264
RL(1), **83**
RN, 605
RNGLR, **390**, 604, 605
robust parsing, 533
root set, **404**, 610
root set grammar, **404**
RPP, **606**
RR(1), **83**
RRP grammar, *see* regular right part grammar
rule selection table, 581
RV grammar, *see* register-vector grammar
Rytter combine node, **469**

Rytter proposal, **460**

S(k)LR(j), **315**
s-grammar, **238**, 357
s-language, 584
S/SL, 506
Sanskrit, 642
SCCs, 311
SDF, 632
SECOND set, **256**
sed, 643
self-embedding, **25**, 627, 628, 633, 638, 641
semantic action, **55**, 254, 264, 634
semantic clause, **54**
Semi-Thue rewriting system, 580
sentence, **16**
sentential form, **16**
sentential tree, **493**
sequence, **6**
set, **6**
severe-C(k), **353**
SGML, 141, 599
shared packed parse forest, 605
shift, **74**
shift item, **207**, 298, 363
shift/reduce conflict, 279, **286**, 578, 630
simple LR(1), 314
simple precedence, **273**, 525, 530, 606, 625
simultaneous tree adjunct grammar, 616
single rule, **113**, 316
skeleton, 548
skeleton grammar, 482, **617**
skeleton parse tree, **271**, 272, 374
SL, **619**
SLA(m)LR(k), 586
SLL(1), 357
SLL(1) (simple LL(1)), **238**
SLR, 573
SLR(k), 85, 586
SLR($k > 1$), **315**
SLR(1), **314**, 364, 382, 571, 588, 592, 607
space requirements, 210, 285, 301, 579
SPLL(1) grammar, **357**
spontaneous input, 305
spontaneously generated look-ahead, 304
SPPF, 605
spurious ambiguity, **63**
spurious error message, **522**, 540, 541
STA grammar, 616
stack, **167**
stack activity reduction, 317
stack alphabet, **167**
stack duplication, 384
stack of stacks, 619
stack shift, **588**
stack-controlling LR parser, **587**
stackability error, **525**
stacking conflict, **588**
stacking error, *see* stackability error
start sentential form, 322
start symbol, **13**
state, **141**
state transition, **142**, 316
station, **281**, 305, 332, 410
strict grammar, **618**
strict syntax generator, **618**
strong operator set, **595**
strong-C(k), **353**
strong-LC(1), **348**
strong-LL(k), **255**, 585
strong-LL(1), 247, **247**, 348, 525, 548
strong-LL-regular grammars, 585
strongly connected component, 97, 311, 587, **630**, 641
Subject-Verb-Object language, 139
submatrix technique, **587**
subsequence, 134
subsequence parsing, 422
subset, 604
subset algorithm, 289, 597
subset construction, **145**, 283, 291, 405
substring parsing, 632
successful production, **47**
suffix grammar, **401**, 540, 626
suffix item, 610
suffix language, **401**
suffix parser, 540, **540**
suffix start set, **407**
supersequence, 134
supersequence parsing, 422, 603
superstring parsing, 422
superstring recognition, **162**
Swiss German, 492
synchronization triplet, **624**
syntactic category, **13**
syntactic graph, *see* production graph
syntax, **7**, 576, 607, 615
syntax error, **521**, 541
Syntax Language, *see* SL
syntaxis, *see* syntax
synthesized attribute, 55, **486**

table compression, 146, 256, 285, 586, 590, 592, 630
table-driven, **70**, 254, 371, 549
tabular parsing, 129, **131**, 509
TAG, 470, **492**
tagged NFA, 599
terminal, **13**, 33
terminal parser, **618**
terminal symbol, **13**
terminal tree, **493**
thread, 86
time complexity, **71**, 500
time requirements, **71**, 579
TMG, 506
TMG recognition Scheme, *see* TS
token, *see* terminal symbol
Tomita parsing, **382**, 601
top-down, 581
top-down parsing, **66**
topological sort, 272, 643
total precedence, 85, **359**, 595, 606
transduction grammar, **55**
transformational grammar, 614
transformations on grammars, 40, 119, 128, 169, 174, 299, 589, 593
transition, 298
transition diagram, **142**, 598, 623, 630
transition graph, **45**
transition network, 641
transitive closure, **96**, 227, 240, 305, 389, 396, 536, 579, 587, 595, 600, 609
transitive item, **226**, 233, 582
tree, **23**, 34
Tree Adjoining Grammars, *see* TAG

trellis automaton, 622
trivial bottom-up table, **381**
trivial top-down table, **381**
TS, **615**
two-level grammar, 73, **477**, 620
Type 0, 19
Type 1 context-sensitive, **20**
Type 1 monotonic, **20**
Type 2.5, 30
Type 3, 30
Type 4, 19, 491
typesetting, 316, 634

undecidable, **36**, 328, 585, 617, 638
undefined non-terminal, 48, 111
undefined terminal, 48, 111
undirected graph, **16**
Unger parser, 85, **104**, 127, 430, 523, 527, 546, 553, 578
union, **52**, 152
uniquely assignable, 617
unit rule, **112**, 121, 147, 316
unreachable non-terminal, 48, 51, 120, 129, 427
unreduce, **201**
unshift, **201**
unsolvable, **36**, 72, 333, 482
unused non-terminal, 48
useless non-terminal, **48**
useless rule, **47**, 48
useless transition, **197**

uvw theorem, 45, 59
uvwxy theorem, 42

Valiant parser, 579
Valiant's algorithm, 98
validation, 540
van Wijngaarden grammars, *see* VW grammars
variable, **13**
VCG, 605
visit, **617**
VW grammar, **477**, 581, 615, 617

weak precedence, **273**, 525, 549, 595
weakly context-sensitive language, **616**
well-formed, 615, 622
well-formed affix grammar, **490**
well-formed substring table, **117**, 546, 580
well-tuned Earley/CYK parser, 224
word, **6**

X-category, 614
XML, 141, 550, 634
XML validation, 148, 598

yacc, 41, 301–303, 311, 316, 633, 634

zero as a number, 41
zip, 567